INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

X.292

(09/98)

SERIES X: DATA NETWORKS AND OPEN SYSTEM COMMUNICATIONS

Open Systems Interconnection – Conformance testing

OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – The Tree and Tabular Combined Notation (TTCN)

ITU-T Recommendation X.292

(Previously CCITT Recommendation)

ITU-T X-SERIES RECOMMENDATIONS

DATA NETWORKS AND OPEN SYSTEM COMMUNICATIONS

PUBLIC DATA NETWORKS	
Services and facilities	X.1–X.19
Interfaces	X.20–X.49
Transmission, signalling and switching	X.50–X.89
Network aspects	X.90–X.149
Maintenance	X.150–X.179
Administrative arrangements	X.180–X.199
OPEN SYSTEMS INTERCONNECTION	
Model and notation	X.200–X.209
Service definitions	X.210–X.219
Connection-mode protocol specifications	X.220–X.229
Connectionless-mode protocol specifications	X.230–X.239
PICS proformas	X.240–X.259
Protocol Identification	X.260–X.269
Security Protocols	X.270–X.279
Layer Managed Objects	X.280–X.289
Conformance testing	**X.290–X.299**
INTERWORKING BETWEEN NETWORKS	
General	X.300–X.349
Satellite data transmission systems	X.350–X.399
MESSAGE HANDLING SYSTEMS	X.400–X.499
DIRECTORY	X.500–X.599
OSI NETWORKING AND SYSTEM ASPECTS	
Networking	X.600–X.629
Efficiency	X.630–X.639
Quality of service	X.640–X.649
Naming, Addressing and Registration	X.650–X.679
Abstract Syntax Notation One (ASN.1)	X.680–X.699
OSI MANAGEMENT	
Systems Management framework and architecture	X.700–X.709
Management Communication Service and Protocol	X.710–X.719
Structure of Management Information	X.720–X.729
Management functions and ODMA functions	X.730–X.799
SECURITY	X.800–X.849
OSI APPLICATIONS	
Commitment, Concurrency and Recovery	X.850–X.859
Transaction processing	X.860–X.879
Remote operations	X.880–X.899
OPEN DISTRIBUTED PROCESSING	X.900–X.999

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

ITU-T RECOMMENDATION X.292

OSI CONFORMANCE TESTING METHODOLOGY AND FRAMEWORK FOR PROTOCOL RECOMMENDATIONS FOR ITU-T APPLICATIONS – THE TREE AND TABULAR COMBINED NOTATION (TTCN)

Summary

This Recommendation defines an informal test notation, called the Tree and Tabular Combined Notation (TTCN), for OSI conformance test suites, which is independent of test methods, layers and protocols, and which reflects the abstract testing methodology defined in Recommendations X.290 and X.291.

Source

ITU-T Recommendation X.292 was revised by ITU-T Study Group 7 (1997-2000) and was approved under the WTSC Resolution No. 1 procedure on the 25th of September 1998.

FOREWORD

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

The World Telecommunication Standardization Conference (WTSC), which meets every four years, establishes the topics for study by the ITU-T Study Groups which, in their turn, produce Recommendations on these topics.

The approval of Recommendations by the Members of the ITU-T is covered by the procedure laid down in WTSC Resolution No. 1.

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

NOTE

In this Recommendation the term *recognized operating agency (ROA)* includes any individual, company, corporation or governmental organization that operates a public correspondence service. The terms *Administration*, *ROA* and *public correspondence* are defined in the *Constitution of the ITU (Geneva, 1992)*.

INTELLECTUAL PROPERTY RIGHTS

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

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

CONTENTS

			Page
1	Scope		1
2	Normative references		2
3	Definitions		3
	3.1	Basic terms from Recommendation X.290	3
	3.2	Terms from Recommendation X.200	5
	3.3	Terms from Recommendation X.210	5
	3.4	Terms from Recommendation X.680	5
	3.5	Terms from Recommendation X.690	5
	3.6	TTCN specific terms	5
4	Abbreviations		10
	4.1	Abbreviations defined in Recommendation X.290	10
	4.2	Abbreviations defined in Recommendation X.291	10
	4.3	Other abbreviations	10
5	Syntax forms of TTCN		11
6	Compliance		12
7	Conventions		12
	7.1	Introduction	12
	7.2	Syntactic metanotation	12
	7.3	TTCN.GR table proformas	13
	7.4	Free Text and Bounded Free Text	15
8	Concurrency in TTCN		15
	8.1	Test Components	15
	8.2	Test Component Configurations	16
9	TTCN Test Suite structure		17
	9.1	Introduction	17
	9.2	Test Group References	17
	9.3	Test Step Group References	17
	9.4	Default Group References	18
	9.5	Parts of a TTCN test suite	18
10	Test Suite Overview		18
	10.1	Introduction	18
	10.2	Test Suite Structure	19
	10.3	Test Case Index	20
	10.4	Test Step Index	21
	10.5	Default Index	22
	10.6	Test Suite Exports	23
	10.7	Import Part	24
11	Declarations Part		26
	11.1	Introduction	26
	11.2	TTCN types	27
	11.3	TTCN operators and TTCN operations	34
	11.4	Test Suite Parameter Declarations	43
	11.5	Test Case Selection Expression Definitions	43
	11.6	Test Suite Constant Declarations	44
	11.7	Test Suite Constant Declarations by Reference	45
	11.8	TTCN variables	46
	11.9	PCO Type Declaration	48

Page

11.10 PCO Declarations 49
11.11 CP Declarations 51
11.12 Timer Declarations 52
11.13 Test Components and Configuration Declarations 54
11.14 ASP Type Definitions 57
11.15 PDU Type Definitions 61
11.16 Test Suite Encoding Information 66
11.17 CM Type Definitions 71
11.18 String length specifications 73
11.19 ASP, PDU and CM Definitions for SEND events 74
11.20 ASP, PDU and CM Definitions for RECEIVE events 74
11.21 Alias Definitions 74

12 Constraints Part 75
12.1 Introduction 75
12.2 General principles 76
12.3 Parameterization of constraints 76
12.4 Chaining of constraints 77
12.5 Constraints for SEND events 77
12.6 Constraints for RECEIVE events 78

13 Specification of constraints using tables 84
13.1 Introduction 84
13.2 Structured Type Constraint Declarations 84
13.3 ASP Constraint Declarations 86
13.4 PDU Constraint Declarations 86
13.5 Parameterization of constraints 88
13.6 Base constraints and modified constraints 88
13.7 Formal parameter lists in modified constraints 89
13.8 CM Constraint Declarations 89

14 Specification of constraints using ASN.1 90
14.1 Introduction 90
14.2 ASN.1 Type Constraint Declarations 90
14.3 ASN.1 ASP Constraint Declarations 91
14.4 ASN.1 PDU Constraint Declarations 92
14.5 Parameterized ASN.1 constraints 93
14.6 Modified ASN.1 constraints 93
14.7 Formal parameter lists in modified ASN.1 constraints 94
14.8 ASP Parameter and PDU field names within ASN.1 constraints 94
14.9 ASN.1 CM Constraint Declarations 95

15 Dynamic Part 95
15.1 Introduction 95
15.2 Test Case dynamic behaviour 95
15.3 Test Step dynamic behaviour 98
15.4 Default dynamic behaviour 99
15.5 Behaviour description 101
15.6 Tree notation 101
15.7 Tree names and parameter lists 102
15.8 TTCN statements 103
15.9 TTCN test events 103
15.10 Expressions 110
15.11 Pseudo-events 116
15.12 Timer management 116

Page

15.13 ATTACH construct 118
15.14 Labels and the GOTO construct 122
15.15 REPEAT construct 123
15.16 Constraints Reference 123
15.17 Verdicts 124
15.18 Meaning of Defaults 127

16 Page continuation 136
16.1 Page continuation of TTCN tables 136
16.2 Page continuation of dynamic behaviour tables 137

Annex A – Syntax and static semantics of TTCN 138
A.1 Introduction 138
A.2 Conventions for the syntax description 138
A.3 TTCN.MP syntax productions in BNF 139
A.4 General static semantics requirements 167
A.5 Differences between TTCN.GR and TTCN.MP 171
A.6 List of BNF production numbers 171

Annex B – Operational Semantics of TTCN 182
B.1 Introduction 182
B.2 Precedence 182
B.3 Processing of test case errors 182
B.4 Converting a modularized test suite to an equivalent expanded test suite 182
B.5 TTCN operational semantics 184

Annex C – TTCN Modules 205
C.1 Introduction 205
C.2 TTCN Module Overview Part 205
C.3 Import Part 208

Annex D – Test Suite Index 209
D.1 Introduction 209
D.2 The Test Suite Index 210

Annex E – Compact proformas 210
E.1 Introduction 210
E.2 Compact proformas for constraints 211
E.3 Compact proforma for Test Cases 216

Appendix I – Examples 217
I.1 Examples of tabular constraints 217
I.2 Examples of ASN1 constraints 221
I.3 Base and modified constraints 228
I.4 Type definition using macros 230
I.5 Use of REPEAT 231
I.6 Test suite operations 231
I.7 Example of a Test Suite Overview 232
I.8 Example of a Test Case in TTCN.MP Form 234
I.9 Use of Component Reference for Field Value Assignment in Constraints 236
I.10 Multi-Party Testing 237
I.11 Multiplexing/Demultiplexing 238
I.12 Splitting and Recombining 238
I.13 Multi-Protocol Test Cases 238
I.14 Example of Modular TTCN 240
I.15 Example of CREATE and DONE 240

Page

Appendix II – Style guide 245
II.1 Introduction 245
II.2 Test case structure 245
II.3 Use of TTCN with different abstract test methods 246
II.4 Use of Defaults 247
II.5 Limiting the execution time of a Test Case 247
II.6 Structured Types 247
II.7 Abbreviations 248
II.8 Test descriptions 248
II.9 Assignments on SEND events 248
II.10 Multi-service PCOs 248
Appendix III – Index 249
III.1 Introduction 249
III.2 The Index 249

Introduction

This Recommendation defines an informal test notation, called the Tree and Tabular Combined Notation (TTCN), for use in the specification of OSI abstract conformance test suites.

In constructing a standardized abstract test suite, a test notation is used to describe abstract test cases. The test notation can be an informal notation (without formally defined semantics) or a Formal Description Technique (FDT). TTCN is an informal notation with clearly defined, but not formally defined semantics.

TTCN is designed to meet the following objectives:

a) to provide a notation in which abstract test cases can be expressed in standardized test suites;

b) to provide a notation which is independent of test methods, layers and protocols;

c) to provide a notation which reflects the abstract testing methodology defined in X.290-series Recommendations;

d) to provide a capability to use concurrency in the specification of abstract test cases, when appropriate, in both multi-party testing and single-party testing.

In the abstract testing methodology a test suite is looked upon as a hierarchy ranging from complete test suite, through test group, test cases and test steps, down to test events. TTCN provides a naming structure to reflect the positions of test cases in this hierarchy. It also provides the means of structuring test cases as a hierarchy of test steps culminating in test events. In TTCN, the basic test events are sending and receiving Abstract Service Primitives (ASPs), Protocol Data Units (PDUs) and timer events.

Two forms of the notation are provided: a human-readable tabular form, called TTCN.GR for use in OSI conformance test suite standards, and a machine processable form, called TTCN.MP, for use in representing TTCN in a canonical form within computer systems and as the syntax to be used when transferring TTCN test cases between different computer systems. The two forms are semantically equivalent.

Recommendation X.292

OSI CONFORMANCE TESTING METHODOLOGY AND FRAMEWORK FOR PROTOCOL RECOMMENDATIONS FOR ITU-T APPLICATIONS – THE TREE AND TABULAR COMBINED NOTATION (TTCN)[1]

(revised in 1998)

The ITU-T,

considering

a) that Recommendation X.200 defines the Reference Model of Open Systems Interconnection for ITU-T Applications;

b) that the objective of OSI will not be completely achieved until systems can be tested to determine whether they conform to the relevant OSI protocol Recommendations;

c) that standardized test suites should be developed for each OSI protocol Recommendation as a means to:

– obtain wide acceptance and confidence in conformance test results produced by different testers;

– provide confidence in the interoperability of equipments which passed the standardized conformance tests;

d) the need for standardizing the conformance testing process to achieve an acceptable and useful degree of comparability of results of conformance assessments of similar products,

unanimously declares the view

that the notation in which generic and abstract test cases are written should be in accordance with this Recommendation.

1 Scope

1.1 This Recommendation defines an informal test notation, called the Tree and Tabular Combined Notation (TTCN), for OSI conformance test suites, which is independent of test methods, layers and protocols, and which reflects the abstract testing methodology defined in Recommendations X.290 and X.291.

1.2 It also specifies requirements and provides guidance for using TTCN in the specification of system-independent conformance test suites for one or more OSI Recommendations. It specifies two forms of the notation: one, a human-readable form, applicable to the production of conformance test suite Recommendations for OSI protocols, and the other, a machine-processable form, applicable to processing within and between computer systems.

1.3 This Recommendation applies to the specification of conformance test cases which can be expressed abstractly in terms of control and observation of protocol data units and abstract service primitives. Nevertheless, for some protocols, test cases may be needed which cannot be expressed in these terms. The specification of such test cases is outside the scope of this Recommendation, although those test cases may need to be included in a conformance test suite Recommendation.

For example, some static conformance requirements related to an application service may require testing techniques which are specific to that particular application.

The specification of test cases in which more than one behaviour description is to be run in parallel is dealt with by the concurrency features (particularly involving the definition of Test Components and Test Component Configurations).

---

[1] Recommendation X.292 and ISO/IEC 9646-3, Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 3: The Tree and Tabular Combined Notation (TTCN), are technically aligned.

1.4 This Recommendation specifies requirements on what a test suite Recommendation may specify about a conforming realization of the test suite, including the operational semantics of TTCN test suites.

1.5 This Recommendation applies to the specification of conformance test suites for OSI protocols in OSI layers 2 to 7, specifically including Abstract Syntax Notation One (ASN.1) based protocols. The following are outside the scope of this Recommendation:

a) the specification of conformance test suites for Physical layer protocols;

b) the relationship between TTCN and formal description techniques;

c) the means of realization of Executable Test Suites (ETS) from abstract test suites.

1.6 This Recommendation defines mechanisms for using concurrency in the specification of abstract test cases. Concurrency in TTCN is applicable to the specification of test cases:

a) in a multi-party testing context;

b) which handle multiplexing and demultiplexing in either a single-party or multi-party testing context;

c) which handle splitting and recombining in either a single-party or multi-party testing context;

d) in a single-party testing context when the complexity of the protocol or set of protocols handled by the IUT is such that concurrency can simplify the specification of the test case.

1.7 TTCN modules are defined to allow sharing of common TTCN specifications between test suites.

2 Normative references

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; all users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published.

– ITU-T Recommendation X.200 (1994) | ISO/IEC 7498-1:1994, *Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model.*

– ITU-T Recommendation X.210 (1993) | ISO/IEC 10731:1994, *Information technology – Open Systems Interconnection – Basic Reference Model: Conventions for the definition of OSI Services.*

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

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

– ITU-T Recommendation X.291 (1995), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – Abstract test suite specification.*

  ISO/IEC 9646-2:1994, *Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 2: Abstract test suite specification.*

– ITU-T Recommendation X.293 (1995), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – Test realization.*

  ISO/IEC 9646-4:1994, *Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 4: Test realization.*

– ITU-T Recommendation X.294 (1995), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – Requirements on test laboratories and clients for the conformance assessment process.*

ISO/IEC 9646-5:1994, *Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 5: Requirements on test laboratories and clients for the conformance assessment process.*

– ITU-T Recommendation X.295 (1995), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – Protocol profile test specification.*

ISO/IEC 9646-6:1994, *Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 6: Protocol profile test specification.*

– ITU-T Recommendation X.296 (1995), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – Implementation conformance statements.*

ISO/IEC 9646-7:1995, *Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 7: Implementation conformance statements.*

– ITU-T Recommendation X.680 (1997) | ISO/IEC 8824-1:1998, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation.*

– ITU-T Recommendation X.681 (1997) | ISO/IEC 8824-2:1998, *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification.*

– ITU-T Recommendation X.682 (1997) | ISO/IEC 8824-3:1998, *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification.*

– ITU-T Recommendation X.683 (1997) | ISO/IEC 8824-4:1998, *Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications.*

– ITU-T Recommendation X.690 (1997) | ISO/IEC 8825-1:1998, *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).*

– ITU-T Recommendation X.691 (1997) | ISO/IEC 8825-2:1998, *Information technology – ASN.1 encoding rules – Specification of Packed Encoding Rules (PER).*

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

– ISO/IEC 10646-1:1993, *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane.*

3 Definitions

This Recommendation defines the following terms.

3.1 Basic terms from Recommendation X.290

The following terms defined in Recommendation X.290 apply:

a) abstract service primitive;

b) abstract testing methodology;

c) abstract test case;

d) abstract test method;

e) abstract test suite;

f) conformance log;

g) conformance test suite;

h) coordinated test method;

i) distributed test method;

j) executable test case;

k) executable test case error;

l) executable test suite;

m) fail verdict;

n) idle testing state;

o) implementation under test;

p) inconclusive verdict;

q) invalid test event;

r) local test method;

s) lower tester;

t) means of testing;

u) pass verdict;

v) PICS proforma;

w) PIXIT proforma;

x) protocol implementation conformance statement;

y) protocol implementation extra information for testing;

z) point of control and observation;

aa) remote test method;

ab) stable testing state;

ac) standardized abstract test suite;

ad) static conformance requirements;

ae) syntactically invalid test event;

af) system under test;

ag) test body;

ah) test case;

ai) test case error;

aj) test coordination procedures;

ak) test event;

al) test group;

am) test group objective;

an) test laboratory;

ao) test management protocol;

ap) test outcome;

aq) (test) postamble;

ar) (test) preamble;

as) test purpose;

at) test realization;

au) est realizer;

av) test step;

aw) test suite;

ax) test system;

ay) upper tester;

az) (test) verdict;

ba) testing state.

3.2 Terms from Recommendation X.200

The following terms defined in Recommendation X.200 (1995) apply:

a) (N)-layer (particularly for application, session and transport layers);

b) (N)-protocol-data-unit;

c) (N)-service-access-point;

d) subnetwork;

e) transfer syntax.

3.3 Terms from Recommendation X.210

The following terms defined in Recommendation X.210 apply:

– OSI-service-provider.

3.4 Terms from Recommendation X.680

The following terms defined in Recommendation X.680 apply:

a) bitstring type;

b) characterstring type;

c) enumerated type;

d) external type;

e) object identifier;

f) octetstring type;

g) real type;

h) selection type;

i) sequence type;

j) sequence-of type;

k) set type;

l) set-of type;

m) subtype.

NOTE – Where there may be ambiguity with TTCN terms these terms are prefixed with the term ASN.1.

3.5 Terms from Recommendation X.690

The following term defined in Recommendation X.690 applies:

– encoding.

3.6 TTCN specific terms

For the purposes of this Recommendation, the following definitions apply:

3.6.1 applicable encoding rules: The actual encoding rules that are to be used when sending or receiving a PDU, after all relevant encoding defaults and overrides, if any, have been combined.

3.6.2 attach construct: A TTCN statement which attaches a Test Step to a calling tree.

3.6.3 base constraint: Specifies a set of default values for each and every field in an ASP or PDU type definition.

3.6.4 base type: The type from which a type defined in a test suite is derived.

3.6.5 behaviour line: An entry in a dynamic behaviour table representing a test event or other TTCN statement together with associated label, verdict, constraints reference and comment information as applicable.

3.6.6 behaviour tree: A specification of a set of sequences of test events, and other TTCN statements.

3.6.7 blank entry: In a modified compact constraint table, a blank entry in a constraint parameter or field denotes that a constraint value is to be inherited.

3.6.8 calling tree: The behaviour tree to which a Test Step is attached.

3.6.9 compact constraint table: Declaration of a set of constraints for an ASP, PDU or Structured Type arranged in a single table.

3.6.10 compact test case table: Declaration of a set of Test Cases for a given Test Group arranged in a single table.

3.6.11 concurrent test case: A test case which is specified using concurrent TTCN.

3.6.12 concurrent TTCN: TTCN which uses test components and test component configurations in order to express concurrency in the dynamic behaviour of test cases.

3.6.13 constraints part: That part of a TTCN test suite concerned with the specification of the values of ASP parameters and PDU fields being sent to the IUT, and conditions on ASP parameters and PDU fields received from the IUT.

3.6.14 constraints reference: A reference to a constraint, given in a behaviour line.

3.6.15 Coordination Message (CM): An item of structured information which may be transferred from one Test Component to another at a Coordination Point.

3.6.16 Coordination Point (CP): A point within a testing environment, assigned to two Test Components in a Test Component Configuration, where CMs may be exchanged asynchronously between these Test Components.

3.6.17 declarations part: That part of a TTCN test suite concerned with the definition and/or declaration of all non-predefined objects that are used in the test suite.

3.6.18 default behaviour: The events, and other TTCN statements, which may occur at any level of the associated tree, and which are indicated in the Default behaviour proforma.

3.6.19 default group: A named set of default behaviours.

3.6.20 default group reference: A path specifying the logical location of a Default in the Default Library.

3.6.21 default identifier: A unique name for a Default.

3.6.22 default library: The set of the Default behaviours in a test suite.

3.6.23 default reference: A reference to a Default in the Default Library from a Test Case or Test Step table.

3.6.24 derivation path: An identifier, consisting of a base constraint identifier concatenated with one or more modified constraint identifiers, separated by dots and finishing with a dot.

3.6.25 dynamic chaining: The linking from constraint declarations of an ASP parameter or PDU field to the constraint declaration of another PDU by means of parameterization. Which PDUs are chained is specified in the constraints reference of a behaviour line.

3.6.26 dynamic part: That part of a TTCN test suite concerned with the specification of Test Case, Test Step and Default dynamic behaviour descriptions.

3.6.27 expanded test suite: A test suite with all imported objects expanded. This will be a result of converting of a modularized test suite according to the algorithm in Annex B.

3.6.28 explicit external: A named object in the External table. An object which is explicitly declared as external in a module is to be explicitly defined or exported as an external object.

3.6.29 explicitly defined object: An object for which a definition or declaration exists in the module or test suite.

3.6.30 explicitly exported object: A named object in the Exports tables being available for use. If the object is an imported object, the name of the source object is to be given.

3.6.31 explicitly imported object: A named object in the Import tables being available for explicit references.

3.6.32 exported object: An explicitly defined object or explicitly imported object in a source object, made available for use in any other module or test suite. An exported object is either an explicitly exported object or an implicitly exported object.

3.6.33 external object: An object being referred to by its name in a module, but neither imported nor explicitly defined. An external object is to be declared in the External table. An external object may be either explicitly external or implicitly external.

3.6.34 global result variable: A predefined test case variable maintained by a Main Test Component in the MPyT context or by the test case in the SPyT context to record the accumulated effect of all the preliminary results of the test case in order to determine the test verdict.

3.6.35 implicit external: An externally declared object in an export table which is omitted from a corresponding Import table.

3.6.36 implicitly exported object: An explicitly defined object or explicitly imported object, which is not itself explicitly exported but which is referred to by an explicitly exported object.

3.6.37 implicitly imported object: An object referred to by some explicitly imported object. The use of an implicitly imported object is restricted to the explicitly imported objects (from the same source object) referring to it.

3.6.38 implicit send event: A mechanism used in Remote Test Methods for specifying that the IUT should be made to initiate a particular PDU or ASP.

3.6.39 imported object: An object copied from some other source object, being available for use. An imported object is either an explicitly imported object or an implicitly imported object.

3.6.40 level of indentation: Indicates the tree structure of a behaviour description. It is reflected in the behaviour description by indentation of text.

3.6.41 local result variable: A predefined variable maintained by a Test Component to record the accumulated effect of its preliminary results.

3.6.42 local tree: A behaviour tree defined in the same proforma as its calling tree.

3.6.43 Main Test Component (MTC): The single Test Component in a Test Component Configuration responsible for creating and controlling Parallel Test Components and computing and assigning the test verdict.

3.6.44 modified constraint: A constraint defined for an ASP or a PDU that already has a base constraint, and which makes modifications on that base constraint.

3.6.45 modularized test suite: A test suite containing Import tables.

3.6.46 module: A self-contained collection of TTCN objects. All referenced objects are either explicitly defined in the Module, are imported from other sources or are defined as external objects in the module.

3.6.47 non-concurrent test case: A test case which is specified in TTCN but without using concurrent TTCN.

3.6.48 object: An element of one of the object categories listed in A.4.2.2 (for TTCN objects with a globally unique identifier) and A.4.2.6 (for ASN.1 identifiers which are globally unique throughout the test suite).

3.6.49 operational semantics: Semantics explaining the execution of a TTCN behaviour tree.

3.6.50 original source object: The module or test suite where an object is explicitly defined.

3.6.51 otherwise event: The TTCN mechanism for dealing with unforeseen test events in a controlled way.

3.6.52 overview part: That part of a TTCN test suite concerned with presenting an overview of the structure of the test suite, the structure (if any) of the Test Step Library, the structure (if any) of the Default Library and the association of selection expressions (if any) with Test Cases and/or Test Groups. This part also provides indexes to Test Cases, Test Steps and Defaults.

3.6.53 parallel test component [PTC]: A test component created by the main test component.

3.6.54 preliminary result: A result recorded before the end of a test case indicating whether the associated part of the test case passed, failed or was inconclusive.

3.6.55 pseudo-event: A pseudo-event is a TTCN expression or Timer operation appearing on a statement line in the behaviour description without any associated event.

3.6.56 qualified event: An event that has an associated Boolean expression.

3.6.57 receive event: The receipt of an ASP or PDU at a named or implied PCO.

3.6.58 result variable: A predefined test case variable for storing preliminary results. In non-concurrent TTCN there is one result variable called R. In concurrent TTCN, there is one global result variable called R, each PTC has a local result variable called R, and the MTC has a local result variable called MTC_R.

3.6.59 root tree: The main behaviour tree of a Test Case, occurring at the level of entry into the Test Case.

3.6.60 send event: The sending of an ASP or PDU to a named or implied PCO.

3.6.61 set of alternatives: TTCN statements coded at the same level of indentation and belonging to the same predecessor node. They represent the possible events, pseudo-events and constructs which are to be considered at the relevant point in the execution of the Test Case.

3.6.62 single constraint table: A declaration of a constraint for a single ASP or PDU of a given type arranged in a single table.

3.6.63 snapshot semantics: A semantic model to eliminate the effect of timing on the execution of a Test Case, defined in terms of snapshots of the test environment, during which the environment is effectively frozen for a prescribed period.

3.6.64 source object: A module or test suite which is imported and has a corresponding Import table.

3.6.65 specific value: A value in TTCN which does not contain any matching mechanism or unbound variable.

3.6.66 static chaining: The linking from constraint declarations of an ASP parameter or PDU field to the constraint declaration of another PDU by explicitly referencing a constraint as its value.

3.6.67 static semantics: Semantic rules that restrict the usage of the TTCN syntax.

3.6.68 structured type: A collection of one or more ASP parameters or PDU fields which may exist in one or more ASP or PDU type definition which is defined in a separate declaration and which may be used to specify a portion of a flat structure or a substructure within the ASP or PDU.

3.6.69 submodule: A module which is included in another module.

3.6.70 test case identifier: A unique name for a Test Case.

3.6.71 test case variable: One of a set of variables declared globally to the test suite, but whose value is retained only for the execution of a single Test Case.

3.6.72 test component: A named subdivision of a concurrent test case capable of being executed in parallel with other test components, and declared as having a fixed number of PCOs and a fixed or maximal number of CPs.

3.6.73 test component configuration: A fixed arrangement of Test Components, PCOs and CPs that is declared for use in concurrent test cases.

3.6.74 test group reference: A path specifying the logical location of a Test Case in the ATS structure.

3.6.75 test step group: A named set of test steps.

3.6.76 test step group reference: A path specifying the logical location of a Test Step in the Test Step Library.

3.6.77 test step identifier: A unique name for a Test Step.

3.6.78 test step library: The set of the Test Step dynamic behaviour descriptions in the test suite, that are not local Test Steps.

3.6.79 test step objective: An informal statement of what the Test Step is meant to accomplish.

3.6.80 test suite constant: One of a set of constants, not derived from the PICS or PIXIT, which will remain constant throughout the test suite.

3.6.81 test suite parameter: One of a set of constants derived from the PICS or PIXIT which globally parameterize a test suite.

3.6.82 test suite variable: One of a set of variables declared globally to the test suite, and which retain their values between Test Cases.

3.6.83 timeout event: An event which is used within a behaviour tree to check for expiration of a specified timer.

3.6.84 tree attachment: The method of indicating that a behaviour tree specified elsewhere (either at a different point in the current proforma, or as a Test Step in the Test Step Library) is to be included in the current behaviour tree.

3.6.85 tree header: An identifier for a local tree followed by an optional list of formal parameters for the tree.

3.6.86 tree identifier: A name identifying a local tree.

3.6.87 tree leaf: A TTCN statement in a behaviour tree or Test Step which has no specified subsequent behaviour.

3.6.88 tree node: A single TTCN statement.

3.6.89 tree notation: The notation used in TTCN to represent Test Cases as trees.

3.6.90 TTCN statement: An event, a pseudo-event or construct which is specified in a behaviour description.

3.6.91 unforeseen test event: A test event which has not been identified as a test event within a foreseen test outcome in the test suite. It is normally handled using the OTHERWISE event.

3.6.92 unqualified event: An event that does not have an associated Boolean expression.

4 Abbreviations

This Recommendation uses the following abbreviations.

4.1 Abbreviations defined in Recommendation X.290

For the purposes of this Recommendation, the following abbreviations defined in clause 4/X.290, apply:

ASP Abstract Service Primitive

ATS Abstract Test Suite

ETS Executable Test Suite

IUT Implementation under Test

LT Lower Tester

LTCF Lower Tester Control Function

MOT Means of Testing

PCO Point of Control and Observation

PICS Protocol Implementation Conformance Statement

PIXIT Protocol Implementation Extra Information for Testing

SUT System under Test

TMP Test Management Protocol

UT Upper Tester

UTCF Upper Tester Control Function

4.2 Abbreviations defined in Recommendation X.291

For the purposes of this Recommendation, the following abbreviations defined in clause 4/X.291, apply:

DS Distributed Single-layer (test method)

LS Local Single-layer (test method)

RS Remote Single-layer (test method)

TTCN Tree and Tabular Combined Notation

4.3 Other abbreviations

For the purposes of this Recommendation, the following abbreviations also apply:

ASN.1 Abstract Syntax Notation One

BNF the extended Backus-Naur Form used in TTCN

CM Coordination Message

CP Coordination Point

FDT Formal Description Technique

FIFO First in First out

MTC Main Test Component

OSI Open Systems Interconnection

PDU	Protocol Data Unit
PTC	Parallel Test Component
SAP	Service Access Point
TCP	Test Coordination Procedures
TTCN.GR	Tree and Tabular Combined Notation, Graphical form
TTCN.MP	Tree and Tabular Combined Notation, Machine Processable form

5 Syntax forms of TTCN

TTCN is provided in two forms:

– a graphical form (TTCN.GR) suitable for human readability;

– a machine processable form (TTCN.MP) suitable for transmission of TTCN descriptions between machines and possibly suitable for other automated processing.

TTCN.GR is defined using tabular proformas. TTCN.MP is defined using syntax productions which have special TTCN.MP keywords as terminal symbols instead of the fixed parts of the tabular proformas (e.g. the box lines and headers). The entries within the TTCN.GR tables are defined by syntax productions which do not include any TTCN.MP keywords; these productions are common to both TTCN.GR and TTCN.MP.

The syntax productions of TTCN.MP are specified in Annex A. As an aid to clarifying the TTCN.GR description, many of the syntax productions that are common to both TTCN.MP and TTCN.GR are embedded in the text of the body of this Recommendation; these are marked: SYNTAX DEFINITION. To aid readability, some productions will appear in several places in the text.

The syntax productions embedded within the text are intended to be identical copies of the corresponding productions from Annex A, but if there is any conflict Annex A shall take precedence.

The text description of TTCN.GR is intended to be consistent with the underlying syntax as defined in the TTCN.MP syntax productions, except for the differences identified in A.5 and the static semantic restrictions specified in Annex A (which are common to both TTCN.MP and TTCN.GR).

If there is any conflict between the TTCN.GR syntax, on the one hand, and the static and operational semantics, on the other, as described by the text and as described by Annex A, then:

a) except for the differences specified in A.5, the TTCN.MP syntax productions shall have precedence over the text and syntax productions in the body of this Recommendation;

b) the static semantics restrictions specified in A.4 and in the static semantics comments (marked STATIC SEMANTICS) on the syntax productions in A.3 specify restrictions on what is valid TTCN, restricting what is allowed according to the syntax productions;

c) similarly, the operational semantics restrictions specified in the operational semantics comments (marked OPERATIONAL SEMANTICS) on the syntax productions in A.3 specify restrictions on what is valid TTCN at run-time, restricting what is allowed according to the syntax productions;

d) the static and operational semantics restrictions specified in Annex A shall have precedence over the text in the body of this Recommendation.

If an ATS is specified in TTCN.GR in compliance with this Recommendation, then there is a unique corresponding TTCN.MP representation of that ATS sharing the same underlying syntax. These two representations have identical operational semantics. Two different representations of an ATS are equivalent if and only if they have identical operational semantics.

NOTE – If there is a standardized ATS specified in TTCN.GR and an apparently equivalent TTCN.MP representation, but there is a conflict in interpretation of the operational semantics of the two, then the operational semantics of the TTCN.GR takes precedence, because it is the TTCN.GR version that is the standardized ATS.

6 Compliance

6.1 ATSs that comply with this Recommendation shall satisfy the requirements for either TTCN.GR or TTCN.MP.

NOTE – See clause 10/X.290, for an explanation of the use of the term "compliance" in X.290-series Recommendations.

6.2 ATSs that comply with the requirements of TTCN.GR shall satisfy the TTCN.GR syntax requirements stated in clauses 9 through 16 and A.4.

6.3 ATSs that comply with the requirements of TTCN.MP shall satisfy the TTCN.MP syntax requirements stated in A.3.

6.4 ATSs that comply with this Recommendation shall satisfy the static semantic requirements specified in clauses 7 through 16 and Annex A and have operational semantics in accordance with the definition of the operational semantics in Annex B together with the operational semantics restrictions specified in A.3, such that they are semantically valid.

6.5 A standardized ATS that complies with this Recommendation shall require that any realization of that test suite that claims to conform to that standardized ATS shall:

a) have operational semantics equivalent to the operational semantics of the test suite as defined by Annex B;

b) meet the additional operational semantics requirements specified in A.3;

c) comply with Recommendation X.293.

NOTE – If, during execution of the executable test case that conforms to the TTCN specification of the corresponding abstract test case, a static semantic or operational semantic error is detected, then a test laboratory complying with Recommendation X.294 will record an abstract or executable test case error, depending on where the error is located.

7 Conventions

7.1 Introduction

The following conventions have been used when defining the TTCN.GR table proformas and the TTCN.MP grammar.

7.2 Syntactic metanotation

Table 1 defines the metanotation used to specify the extended BNF grammar for TTCN (henceforth called BNF).

Table 1/X.292 – The TTCN.MP Syntactic Metanotation

::=	Is defined to be
abc xyz	abc followed by xyz
|	Alternative
[abc]	0 or 1 instances of abc
{abc}	0 or more instances of abc
{abc}+	1 or more instances of abc
(...)	Textual grouping
abc	The non-terminal symbol abc
abc	A terminal symbol abc
"abc"	A terminal symbol abc

EXAMPLE 1 – Use of the BNF metanotation:

FormalParList ::= "(" FormalPar&Type {SemiColon FormalPar&Type} ")"

The following conventions will be used for text used in table proformas:

a) Bold text (**like this**) shall appear verbatim in each actual table in a TTCN test suite.

b) Text in italics (*like this*) shall not appear verbatim in a TTCN test suite. This font is used to indicate that actual text shall be substituted for the italicized symbol. Syntax requirements for the actual text can be found in the corresponding TTCN.MP BNF production.

EXAMPLE 2 – *SuiteIdentifier* corresponds to production 3 in Annex A.

7.3 TTCN.GR table proformas

7.3.1 Introduction

The TTCN.GR is defined using two types of table:

a) single TTCN object tables (see 7.3.2),

which are used to define, declare or describe a single TTCN object such as a PDU declaration or a Test Case dynamic behaviour;

b) multiple TTCN object tables (see 7.3.3),

are used to define a number of TTCN object of the same type in a single table, such as simple type definitions or Test Case Variables.

7.3.2 Single TTCN object tables

The general lay-out of a table for a single TTCN object is shown in Figure 1.

Title of Table			
Object Name	**:**		
Group	**:** (Optional way of grouping together related objects)		
	: **:**		
Comments	**:** This entire comment line is optional.		
Object Name	**... Other Columns ...**	**Comments**	
		This column is optional	
Detailed Comments:	This footer is optional.		

Title
Header
Body
Footer

Figure 1/X.292 – Generalized layout of a single declaration table

The header of the table contains general information on the object defined in the table. The first item in the header, named *Object Name*, contains an identifier for the object. A second item, named *Group*, may be used to provide an identifier to group together related objects in the same category. This item is optional. The last item, named *Comments* contains an informal description of the object. This item is optional.

The body of the table consists of one or more columns. Each column has a title. The rightmost column, titled *Comments*, contains informal descriptions of the components of the object specified in the body. It does not exist in all proformas. In proformas containing a comments column this column can be omitted.

The footer of the table contains one item, named *Detailed Comments*. This footer can be used for the same purposes as the comments column in the body of the table. The test suite specifier can use the detailed comments footer in combination with the comments column, instead of a comments column, or not at all, in which case the footer can be omitted.

7.3.3 Multiple TTCN object tables

The general lay-out of a table for multiple TTCN objects is shown in Figure 2.

The optional *Collective Comments* may be used preceding a group of related objects declared in a multiple object table, both to indicate the grouping and to give a comment that applies to each member of the group or the group as a whole.

This type of table has only a minimal optional header section, which may contain a *Group* identifier and a *Collective Comment*. The body of the table consists of one or more columns. Each column has a title. The leftmost column, titled *Object Name*, contains identifiers of the objects defined or declared in the table. The rightmost column, titled *Comments*, contains informal descriptions of the objects defined or declared in the table. It does not exist in all proformas. When it exists its use is optional for the test suite specifier. The footer of the table is identical to the footer of the single table type.

Title of Table		
Group : (Optional way of grouping together related sets of objects)		
Collective Comment: *A comment valid for the below defined/declared objects. This comment has a scope reaching to next Collective Comment or until the end of this table.*		
Object Name	**... Other Columns ...**	**Comments**
Collective Comment: *A comment valid for the below defined/declared objects. This comment has a scope reaching to next Collective Comment or until the end of this table.*		
Object Name	**... Other Columns ...**	**Comments**
Detailed Comments:		

Figure 2/X.292 – Generalized layout of a multiple declaration table

7.3.4 Alternative compact tables

In some cases it is allowed to display a number of single TTCN object tables in an alternative space-saving compact format. That is, a number of single TTCN object tables may be displayed in a single compact table. The only tables that may be presented in this format are:

– ASP constraints (tabular and ASN.1);

– PDU constraints (tabular and ASN.1);

– Structured Type constraints;

– ASN.1 Type constraints;

– Test Case dynamic behaviours.

The formats of these alternative compact proformas are defined in Annex E.

7.3.5 Specification of proformas

This Recommendation specifies numerous types of TTCN.GR tables and provides a graphic view of the corresponding proformas. These proformas conform to the generalized lay-out of 7.3.2 and 7.3.3. When a column is shaded in a proforma, this is a reminder that the column is optional.

7.4 Free Text and Bounded Free Text

Some table entries allow the use of free text, i.e. characters from any of the character sets defined in ISO/IEC 10646-1. The following restrictions apply:

a) Free Text shall not contain the combination of characters "*/", unless preceded by backslash (\ ), as this is used in the TTCN.MP to indicate the end of a Free Text string. This means that double backslash (\\ ) means backslash.

b) The combinations of characters "/*" and "*/" which open and close BoundedFreeText strings in the TTCN.MP shall not appear in the TTCN.GR, i.e. wherever a Bounded FreeText string appears in a table section, as in a Full Identifier, these combinations of characters shall not be printed.

8 Concurrency in TTCN

8.1 Test Components

TTCN allows the specification of test components which may be executed concurrently. This clause gives an overview of the additional proformas and mechanisms available in concurrent TTCN. These proformas and mechanisms shall not be used in ATSs that do not use concurrency (i.e. the use of concurrency is optional).

A tester consists of a Main Test Component (MTC) and zero or more Parallel Test Components (PTCs). In non-concurrent TTCN it is not necessary to declare the Main Test Component since there is only one test component and the default is that it is the Main Test Component.

Test components are declared in the Test Component Declarations table. A test component may communicate with the IUT via one or more Points of Control and Observation (PCOs). Test components may communicate with each other by exchanging Coordination Messages (CMs) through Coordination Points (CPs). PTCs may also communicate with the MTC implicitly, by means of assignments to the global result variable and by the MTC being able to check whether or not one or more PTCs have terminated execution. The Test Component Configuration Declarations tables are used to specify (abstract) configurations of test components. These declarations (one for each configuration) show which PCOs and CPs are used, if any, by the test components. CMs are specified in a manner very similar to the method used to specify ASPs. ASN.1 may be used for CM specification. CM constraints are also very similar to ASP constraints. Special proformas are provided for the definition of CM Types and the declaration of CM constraints. CMs are sent and received using the normal TTCN SEND and RECEIVE statements.

In summary, if concurrent TTCN is used, the following proformas shall be used:

a) Test Component Declarations;

b) Test Component Configuration Declarations.

In addition, if concurrent TTCN is used, the following proformas may be used:

c) CP Declarations;

d) CM Type Definitions and/or ASN.1 CM Type Definitions, provided that CP declarations are used;

e) CM Constraints Declarations, provided that CM Type Definitions are used;

f) ASN.1 CM Constraint Declarations, provided that ASN.1 CM Type Definitions are used.

8.2 Test Component Configurations

Some possible configurations of test components are shown in Figures 3 and 4. In a realization of these abstract configurations, test components may reside in a single machine or be distributed over several machines.

It is possible to use different PTC configurations in different test cases of an Abstract Test Suite. Each Abstract Test Case which uses concurrency shall use one of the declared Test Component Configurations.

Note the following valid but unusual cases:

a) A PTC need not have any PCOs.

b) A PTC need not have a CP to an MTC. In such cases the only interaction between the PTC and the MTC will be the creation of the PTC and the implicit result reports from the PTC, i.e. the MTC has no explicit control over the PTC after creation.

c) Two PTCs may be connected by more than one CP.

d) A test case whose test component configuration refers to a PTC need not contain any CREATE statement to start this PTC.

e) A test case whose test component configuration refers to a CP need not contain any SEND or RECEIVE statements using this CP.

Items a), b) and c) are illustrated in Figures 3 and 4.

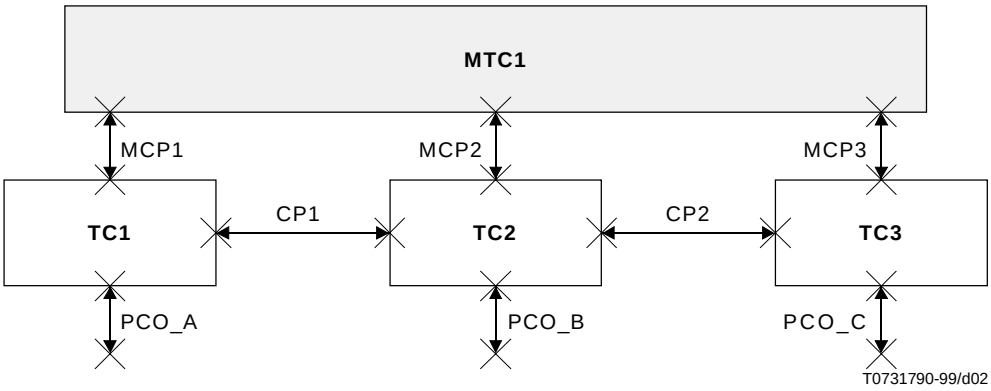


Figure 3/X.292 – Example Test Component Configuration CONFIG1

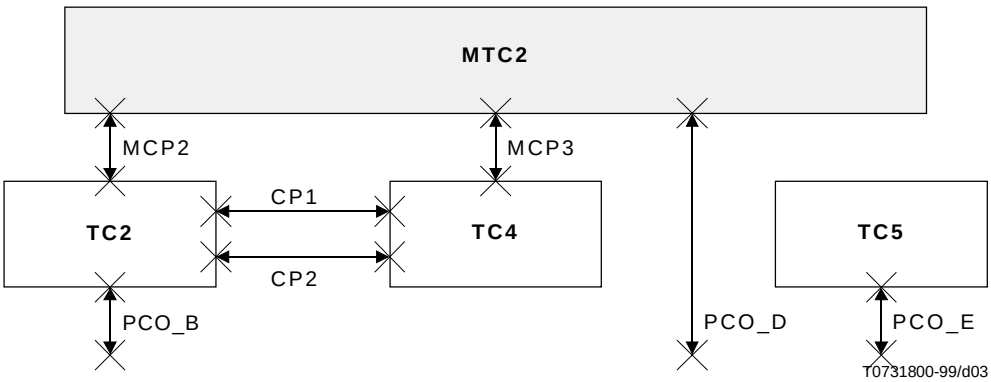


Figure 4/X.292 – Example Test Component Configuration CONFIG2

9 TTCN Test Suite structure

9.1 Introduction

TTCN allows a test suite to be hierarchically structured in accordance with 8.1/X.290. The components of this structure are:

a) Test Groups;

b) Test Cases;

c) Test Steps.

A TTCN test suite may be completely flat (i.e. have no structure) in which case there are no Test Groups.

TTCN allows the use of Test Step Groups and Default Groups, similar to the concept of Test Groups, in order to structure Test Steps and Defaults hierarchically. This hierarchical structure is optional.

9.2 Test Group References

TTCN supports a naming structure that shows a conceptual grouping of Test Cases. Test Groups can be nested. Test Cases can also be stand alone (see Figure 9/X.290). The Test Group References define the structure of the test suite. Test Group References shall have the following syntax:

SYNTAX DEFINITION:

626 TestGroupReference ::= [SuiteIdentifier "/"] {TestGroupIdentifier "/"}

EXAMPLE 3 – A Transport group reference: TRANSPORT/CLASS0/CONN_ESTAB/

9.3 Test Step Group References

9.3.1 Test steps may be explicitly identified in TTCN and used to structure Test Cases and other Test Steps. Alternatively Test Steps may be implicit within the behaviour description of a Test Case. Explicit Test Steps may be specified either:

– locally within a Test Case or Test Step behaviour description; or

– globally within a Test Step Library, which may be hierarchically structured into Test Step Groups.

NOTE – For example, a preamble may consist of just a few statement lines within a behaviour description of the Test Case, in which case it is implicit. Alternatively, a preamble may be explicitly specified with its own behaviour description. If such an explicit preamble is only of use within one Test Case, then it may be specified locally within that Test Case, but if it is of use in several Test Cases, then it should be specified in the Test Step Library.

9.3.2 Local Test Steps are identified simply by a tree identifier. Global Test Steps are identified by a Test Step identifier. Global Test Steps also have a Test Step Group Reference, which shows the position of a Test Step in the Test Step Library. The structure of the Test Step Library is independent of the structure of the test suite. Test Step Group References shall have the following syntax:

SYNTAX DEFINITION:

641 TestStepGroupReference ::= [SuiteIdentifier "/"] {TestStepGroupIdentifier "/"}

EXAMPLE 4 – Transport Test Step Group Reference: TRANSPORT/STEP_LIBRARY/CLASS0/CONN_ESTAB/

9.4 Default Group References

Default behaviours (if any) are located in a Default Library.

A Default Group Reference specifies the location of the Default in the Default Library, which may be hierarchically structured. The Default Library has no influence on the test suite structure itself. Default Group References shall have the following syntax:

SYNTAX DEFINITION:

651 DefaultGroupReference ::= [SuiteIdentifier "/"] {DefaultGroupIdentifier "/"}

EXAMPLE 5 – Transport Default Group Reference: TRANSPORT/DEFAULT_LIBRAR/CLASS0/

9.5 Parts of a TTCN test suite

An ATS written in TTCN shall have the following four sections in the order indicated:

a) Suite Overview (see clause 10),

which contains the information needed for the general presentation and understanding of the test suite, such as test references and a description of its overall purpose;

b) Import Part (see 10.7),

which contains the declarations of the objects used in the test suite or module that are imported from a source object;

c) Declarations Part (see clause 11),

which contains the definitions or declarations of all the components that comprise the test suite (e.g. PCOs, Timers, ASPs, PDUs, and their parameters or fields);

d) Constraints Part (see clauses 12, 13 and 14),

which contains the declarations of values for the ASPs, PDUs, and their parameters used in the Dynamic Part. The constraints shall be specified using:

1) TTCN tables; or

2) the ASN.1 value notation; or

3) both TTCN tables and the ASN.1 value notation;

e) Dynamic Part (see clause 15),

which comprises three sections that contain tables specifying test behaviour expressed mainly in terms of the occurrence of ASPs or PDUs at PCOs. These sections are:

1) the Test Case dynamic behaviour descriptions;

2) a library containing Test Step dynamic behaviour descriptions (if any);

3) a library containing Default dynamic behaviour descriptions (if any).

10 Test Suite Overview

10.1 Introduction

The purpose of the Test Suite Overview part of the ATS is to provide information needed for general presentation and understanding of the test suite. This includes:

a) Test Suite Structure (see 10.2);

b) Test Case Index (see 10.3);

c) Test Step Index (see 10.4);

d) Default Index (see 10.5);

e) Test Suite Exports (see 10.6).

10.2 Test Suite Structure

The Test Suite Structure contains identification of the pertinent reference documents, specification of the structure of the test suite, a brief description of its overall purpose, and references to the Test Group selection criteria.

The Test Suite Structure shall include at least the following information:

a) the name of the test suite;

b) references to the relevant base standards;

c) a reference to the PICS proforma;

d) a reference to the partial PIXIT proforma (see 14.1/X.291, and Appendix V/X.296);

e) an indication of the test method or methods to which the test suite applies, plus for the Coordinated Test Methods a reference to where the TMP is specified;

f) other information which may aid understanding of the test suite, such as its version number or how it has been derived; this information should be included as a comment;

g) a list of Test Groups in the test suite (if any),

   where the following information shall be supplied for each group:

   1) The Test Group Reference,

      where the first identifier may be the suite name, and each successive identifier represents further conceptual ordering of the test suite. Test Groups shall be listed in the order that their corresponding Test Cases appear in the ATS. Furthermore, they shall be ordered such that every group within a single group immediately follows that group. All Test Groups in the test suite shall be listed.

      Imported test cases may be included under any group, independently under which group they are defined in the original source object. A new group may be listed that does not occur in the Dynamic Part. This group shall only contain imported test cases.

      The groups of the Dynamic Part shall occur in the same order as they appear there, but the list may be preceded, interrupted or followed by new groups of imported test cases. For these new groups the page number shall not be supplied.

      The Selection Ref column may contain the identifier of a selection expression applicable to the new test groups. The new selection expression shall override the specified selection expression in the original test group (if there is any). The absence of the selection expression identifier in this column indicates that the specified selection expression in the original test group is omitted (if there is any).

      The Test Group Objective column may contain a new informal statement of the objective of the new test group. This new objective shall override the objective in the imported test group (if any). The absence of the test group objective in this column indicates that the specified test group objective is omitted.

   2) An optional selection expression identifier,

      which references an entry in the Test Case Selection Expression Definitions table used to determine if the Test Cases in the group apply to specific IUTs. This column may contain the identifier of a selection expression applicable to the Test Group. If a selection expression identifier is provided for a group, and the referenced selection expression evaluates to FALSE, then no Test Case in that group shall be selected for execution. If the selection expression evaluates to TRUE, then Test Cases in that group shall be selected for execution depending on the evaluation of the selection expressions relevant to subgroups of that group and/or individual Test Cases. Omission of a selection expression identifier is equivalent to the Boolean value TRUE.

   3) The Test Group Objective,

      which is an informal statement of the objective of the Test Group.

   4) A page number,

      providing the location of the first Test Case of the group in the ATS. The page number listed with each Test Group Reference in the Test Suite Structure table shall be the page number of the first Test Case behaviour description in the group.

This information shall be provided in the format shown in Proforma 1, below.

Test Suite Structure			
Suite Name : *SuiteIdentifier*			
Standards Ref : *Free Text*			
PICS Ref : *Free Text*			
PIXIT Ref : *Free Text*			
Test Method(s) : *FreeText*			
Comments : *[FreeText]*			
Test Group Reference	**Selection Ref**	**Test Group Objective**	**Page No.**
⋮ *TestGroupReference* ⋮	⋮ *[SelectExprIdentifier]* ⋮	⋮ *FreeText* ⋮	⋮ *Number* ⋮
Detailed Comments: *[FreeText]*			

Proforma 1 – Test Suite Structure

SYNTAX DEFINITION:

41 SuiteIdentifier ::= Identifier

626 TestGroupReference ::= [SuiteIdentifier "/"] {TestGroupIdentifier "/"}

202 SelectExprIdentifier ::= Identifier

741 Number ::= (NonZeroNum {Num}) | 0

10.3 Test Case Index

The Test Case Index contains a complete list of all Test Cases in the ATS. The following information shall be provided for each Test Case:

a) An optional Test Group Reference (if the ATS is structured into Test Groups),

which defines where in the test suite group structure the Test Case resides. If the group reference for a Test Case is missing, then the Test Case is assumed to reside in the same Test Group as the previous Test Case in the index. Test Groups shall be listed in the order in which they exist in the ATS. An explicit Test Group Reference shall be provided for the first Test Case of each group. An explicit Test Group Reference shall also be provided for each Test Case that immediately follows the last Test Case of the Test Group; this is necessary if a Test Group contains both Test Groups and Test Cases.

b) The Test Case name,

which shall be the identifier provided in the Test Case dynamic behaviour table. Test Cases shall be listed in the order in which they exist in the ATS.

c) An optional selection expression identifier,

which references an entry in the Test Case Selection Expression Definitions table used to determine if the Test Case should be selected for execution. This column may contain the identifier of a selection expression applicable to the Test Case. If a selection expression identifier is provided, and the referenced selection expression evaluates to FALSE, then the Test Case shall not be selected for execution. If the selection expression evaluates to TRUE, then the Test Case shall be selected for execution depending on the evaluation of the selection expressions for the Test Groups containing the Test Case. A Test Case is selected if the selection expression for the Test Case, and all groups containing the Test Case, evaluate to TRUE. Omission of a selection expression identifier is equivalent to the Boolean value TRUE.

d) A description of the Test Case,

which is possibly a shortened form of the test purpose.

e) A page number,

providing the location of the Test Case in the ATS. The page number listed with each Test Case Identifier in the Test Case Index table shall be the page number of the corresponding Test Case behaviour description.

This information shall be provided in the format shown in Proforma 2, below.

Test Case Index				
Test Group Reference	**Test Case Id**	**Selection Ref**	**Description**	**Page No.**
⋮ *TestGroupReference* ⋮	⋮ *TestCaseIdentifier* ⋮	⋮ *[SelectExprIdentifier]* ⋮	⋮ *FreeText* ⋮	⋮ *Number* ⋮
Detailed Comments: *[FreeText]*				

Proforma 2 – Test Case Index

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

626 TestGroupReference ::= [SuiteIdentifier "/"] {TestGroupIdentifier "/"}

624 TestCaseIdentifier ::= Identifier

202 SelectExprIdentifier ::= Identifier

The complete list of test cases shall include the imported test cases. Explicitly defined Test Cases shall be listed in the order in which they exist in the ATS. Page numbers shall not be supplied for imported test cases.

The Selection Ref column has similar semantic as the one given in the previous subclause (see 10.2).

The Description column may contain a new shortened form of the Test Purpose. This new description shall override the description in the imported test case (if any). The absence of the description in this column indicates that the specified description is omitted.

10.4 Test Step Index

The Test Step Index contains a complete list of all Test Steps in the ATS. The following information shall be provided for each Test Step:

a) An optional Test Step Group Reference (if the ATS is structured into Test Step Groups),

which defines where in the Test Step Library structure the Test Step resides. If the group reference for a Test Step is missing, then the Test Step is assumed to reside in the same group as the previous Test Step in the index. Test Step Groups shall be listed in the order in which they exist in the ATS. An explicit Test Step Group Reference shall be provided for the first Test Step of each group. An explicit Test Step Group Reference shall also be provided for each Test Step that immediately follows the last Test Step of the group; this is necessary if a Test Step Group contains both Test Step Groups and Test Steps.

b) The Test Step name,

which shall be the identifier provided in the Test Step dynamic behaviour table. Test Steps shall be listed in the order in which they exist in the ATS.

c) A description of the Test Step,

which is possibly a shortened form of the Test Step Objective.

d) A page number,

providing the location of the Test Step in the ATS. The page number listed with each Test Step Identifier in the Test Step Index table shall be the page number of the corresponding Test Step behaviour description.

This information shall be provided in the format shown in Proforma 3, below.

Test Step Index			
Test Step Group Reference	**Test Step Id**	**Description**	**Page No.**
⋮ *TestStepGroupReference* ⋮	⋮ *TestStepIdentifier* ⋮	⋮ *FreeText* ⋮	⋮ *Number* ⋮
Detailed Comments: *[FreeText]*			

Proforma 3 – Test Step Index

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

641 TestStepGroupReference ::= [SuiteIdentifier "/"] {TestStepGroupIdentifier "/"}

639 TestStepIdentifier ::= Identifier

The complete list of test steps shall include the imported test steps. Explicitly defined Test Steps shall be listed in the order in which they exist in the ATS. Page numbers shall not be supplied for imported test steps.

The Description column may contain a new shortened form of the Test Step Objective. This new description shall override the description in the imported test step (if any). The absence of the description in this column indicates that the specified description is omitted.

10.5 Default Index

The Default Index contains a complete list of all Defaults in the ATS. The following information shall be provided for each Default:

a) An optional Default Group Reference, (if the ATS is structured into Default Groups),

which defines where in the Default Library structure the Default resides. If the group reference for a Default is missing, then the Default is assumed to reside in the same group as the previous Default in the index. Defaults shall be listed in the order in which they exist in the ATS. An explicit Default Group Reference shall be provided for the first Default of each group. An explicit Default Group Reference shall also be provided for each Default that immediately follows the last Default of the group.

b) The Default name,

which shall be the identifier provided in the Default dynamic behaviour table. Defaults shall be listed in the order in which they exist in the ATS.

c) A description of the Default,

which is possibly a shortened form of the Default Objective.

d) A page number,

providing the location of the Default in the ATS. The page number listed with each Default Identifier in the Default Index table shall be the page number of the corresponding Default behaviour description.

This information shall be provided in the format shown in Proforma 4, below.

Default Index			
Default Group Reference	**Default Id**	**Description**	**Page No.**
⋮ *DefaultGroupReference* ⋮	⋮ *DefaultIdentifier* ⋮	⋮ *FreeText* ⋮	⋮ *Number* ⋮
Detailed Comments: *[FreeText]*			

Proforma 4 – Default Index

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

651 DefaultGroupReference ::= [SuiteIdentifier "/"] {DefaultGroupIdentifier "/"}

650 DefaultIdentifier ::= Identifier

The complete list of defaults shall include the imported defaults. Explicitly defined Defaults shall be listed in the order in which they exist in the ATS. Page numbers shall not be supplied for imported defaults.

The Description column may contain a new shortened form of the Default Objective. This new description shall override the description in the imported default (if any). The absence of the description in this column indicates that the specified description is omitted.

10.6 Test Suite Exports

The Test Suite Exports table may be used to specify explicitly which objects in the test suite are designed to be re-usable and hence may be imported into other test suites or TTCN modules.

The Test Suite Exports proforma is used to identify the objects which may be exported.

If a PCO type is given as an exported object in the Export table, it shall be defined in the optional PCO Type table.

The name of the original source object shall be given if the object is itself imported.

If the object is declared as an external object (explicit external) or is an object which is omitted in the imported source object (implicit external), the keyword EXTERNAL is given instead of the source object name.

Exporting an object of type Enumeration or Named Number requires that the corresponding type is given. The other objects which are defined in the corresponding type are not exported as well. They are however implicitly exported and can be referred in other exported objects. The type name is given as a suffix to the object name embedded in brackets.

The following information shall be supplied in the Test Suite Exports table for each of the exported objects:

a) The name of the object – If the object is of type NamedNumber or Enumeration, the corresponding type shall be given as a suffix to the object name embedded in brackets.

b) The object type.

c) The name of the original source object if the object is imported, or the object directive EXTERNAL.

d) A page number,

   providing the location of the object in the test suite (no page number shall be given for imported objects).

e) An optional comment.

This information shall be provided in the format shown in Proforma 5, below.

Test Suite Exports				
Object Name	**Object Type**	**Source Name**	**Page No.**	**Comments**
⋮ *ObjectIdentifier* ⋮	⋮ *TTCN_ObjectType* ⋮	⋮ *[SourceIdentifier | ObjectDirective]* ⋮	⋮ *Number* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*				

Proforma 5 – Test Suite Exports

SYNTAX DEFINITION:

12 ObjectIdentifier ::= Identifier | ObjectTypeReference

15 TTCN_ObjectType ::= **SimpleType_Object** | **StructType_Object** | **ASN1_Type_Object** | **TS_Op_Object** | **TS_Proc_Object** | **TS_Par_Object** | **SelectExpr_Object** | **TS_Const_Object** | **TS_Var_Object** | **TC_Var_Object** | **PCO_Type_Object** | **PCO_Object** | **CP_Object** | **Timer_Object** | **TComp_Object** | **TCompConfig_Object** | **TTCN_ASP_Type_Object** | **ASN1_ASP_Type_Object** | **TTCN_PDU_Type_Object** | **ASN1_PDU_Type_Object** | **TTCN_CM_Type_Object** | **ASN1_CM_Type_Object** | **EncodingRule_Object** | **EncodingVariation_Object** | **InvalidFieldEncoding_Object** | **Alias_Object** | **StructTypeConstraint_Object** | **ASN1_TypeConstraint_Object** | **TTCN_ASP_Constraint_Object** | **ASN1_ASP_Constraint_Object** | **TTCN_PDU_constraint_Object** | **ASN1_PDU_Constraint_Object** | **TTCN_CM_Constraint_Object** | **ASN1_CM_Constraint_Object** | **TestCase_Object** | **TestStep_Object** | **Default_Object** | **NamedNumber_Object** | **Enumeration_Object**

17 SourceIdentifier ::= SuiteIdentifier | TTCN_ModuleIdentifier

18 ObjectDirective ::= Omit | EXTERNAL

EXAMPLE 6 – Test Suite Exports:

Test Suite Exports				
Object Name	**Object Type**	**Source Name**	**Page No.**	**Comments**
String5	SimpleTypeDef		3	
wait	TimerDcl	Module_B		
INTC	TTCN_PDU_Type		13	
DEF1	Default	TestSuite_1		
TC_2	TestCase	TestSuite_2		
TC_3	TestCase		33	
Preamble	TestStep	EXTERNAL		
Detailed Comments:				

10.7 Import Part

10.7.1 Introduction

The purpose of the Import Part is to declare the objects used in the test suite that are imported from a source object. The effect of the imports is equivalent to having a copy of the imported objects within the test suite.

An object may be imported only if it is exported by a source object. A test suite without an export table exports all objects which have a global name. A module and a test suite with at least one export table export the objects contained in the export tables. An object which is not itself explicitly imported is implicitly imported if it is referenced by an imported object.

10.7.2 Imports

The Imports table identifies the source object and provides information on the overall objective of the source object. The following information shall be supplied in the Imports table:

a) the name of the source object;

b) a description of the objective of the source object;

c) a full reference to the source object; which should contain a document identifier and other information, such as version and date;

d) other information which may aid understanding of the source object, this should be included as a comment;

e) a list of the objects from the imported source object; for each object the following information shall be provided:

   1) the name of the object as used in the source object;

   2) the type of the object; which shall be the same as the type given in the source object;

   3) the name of the original source object if the object is imported from another source object, the object directive OMIT or "-" if the object is to be omitted from the set of objects imported from the source object, or the object directive EXTERNAL if the object is declared as external in the source object.

This information shall be provided in the format shown in Proforma 6, below.

Imports			
Suite Name	**:** *SuiteIdentifier*		
Group	**:** *[ImportsGroupReference]*		
Standards Ref	**:** *[FreeText]*		
Comments	**:** *[FreeText]*		
Object Name	**Object Type**	**Source Name**	**Comments**
⋮	⋮	⋮	⋮
ObjectIdentifier	*TTCN_ObjectType*	*[SourceIdentifier | ObjectDirective]*	*[FreeText]*
⋮	⋮	⋮	⋮
Detailed Comments: *[FreeText]*			

Proforma 6 – Imports

SYNTAX DEFINITION:

17 SourceIdentifier ::= SuiteIdentifier | TTCN_ModuleIdentifier

34 ImportsGroupReference ::=[(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ImportsGroupIdentifier "/"}

12 ObjectIdentifier ::= Identifier | ObjectTypeReference

15 TTCN_ObjectType ::= **SimpleType_Object** | **StructType_Object** | **ASN1_Type_Object** | **TS_Op_Object** | **TS_Proc_Object** | **TS_Par_Object** | **SelectExpr_Object** | **TS_Const_Object** | **TS_Var_Object** | **TC_Var_Object** | **PCO_Type_Object** | **PCO_Object** | **CP_Object** | **Timer_Object** | **TComp_Object** | **TCompConfig_Object** | **TTCN_ASP_Type_Object** | **ASN1_ASP_Type_Object** | **TTCN_PDU_Type_Object** | **ASN1_PDU_Type_Object** | **TTCN_CM_Type_Object** | **ASN1_CM_Type_Object** | **EncodingRule_Object** | **EncodingVariation_Object** | **InvalidFieldEncoding_Object** | **Alias_Object** | **StructTypeConstraint_Object** | **ASN1_TypeConstraint_Object** | **TTCN_ASP_Constraint_Object** | **ASN1_ASP_Constraint_Object** | **TTCN_PDU_constraint_Object** | **ASN1_PDU_Constraint_Object** | **TTCN_CM_Constraint_Object** | **ASN1_CM_Constraint_Object** | **TestCase_Object** | **TestStep_Object** | **Default_Object** | **NamedNumber_Object** | **Enumeration_Object**

18 **ObjectDirective ::= Omit | EXTERNAL**

EXAMPLE 7 – An Imports table:

Imports			
Test Case Name	**:** Module A		
Source Ref	**:** {ISO standard 1234}		
Standards Ref	**:** ISO 300 313		
Comments	**:** Layer 2 Test Suite		
Object Name	**Object Type**	**Source Name**	**Comments**
String5	SimpleTypeDef		
Wait	TimeDcl	Module B	1)
R1_POSTAMBLE	TestStep	EXTERNAL	2)
TSAP	PCO_TypeDcl		3)
blue[ColorEnum]	Enumeration	OMIT	
a[NN_type1]	NamedNumber		4)
Detailed Comments:			
1) The original source of this is Module B.			
2) This test step is declared as external in Module A and must be explicitly defined or imported where this module is used.			
3) TSAP must be defined in the PCO Type Dcl table.			
4) This Named Number is omitted from the imports and hence should be redefined explicitly in the test suite.			

11 Declarations Part

11.1 Introduction

The purpose of the declarations part of the ATS is to define and declare all the objects used in the test suite. The following objects of an ATS referenced from the overview part, the constraints part and the dynamic part shall have been declared in the declarations part. These objects are:

a) *Definitions:*

   1) Test Suite Types (see 11.2.3);

   2) Test Suite Operations (see 11.3.4).

b) *Parameterization and selection of Test Cases:*

   1) Test Suite Parameters (see 11.4);

   2) Test Case Selection Expressions (see 11.5).

c) *Declarations/definitions:*

   1) Test Suite Constants (see 11.6 and 11.7);

   2) Test Suite Variables (see 11.8.1);

   3) Test Case Variables (see 11.8.3);

   4) PCO types (see 11.9);

   5) PCOs (see 11.10);

   6) CPs (see 11.11);

   7) Timers (see 11.12);

   8) Test Components (see 11.13.1);

   9) Test Component Configurations (see 11.13.2);

   10) ASP types (see 11.14);

   11) PDU types (see 11.15);

12) Encoding Rules (see 11.16.1);

13) Encoding Variations (see 11.16.2);

14) Invalid Field Encodings (see 11.16.3);

15) CM types (see 11.17);

16) Aliases (see 11.21).

11.2 TTCN types

11.2.1 Introduction

TTCN supports a number of predefined types and mechanisms that allow the definition of specific Test Suite Types. These types may be used throughout the test suite and may be referenced when Test Suite Parameters, Test Suite Constants, Test Suite Variables, ASP parameters, PDU fields, etc. are declared.

TTCN is a weakly typed language, in that values of any two types which have the same base type are considered to be type compatible (e.g. for the purposes of performing assignments or parameter passing).

11.2.2 Predefined TTCN types

A number of commonly used types are predefined for use in TTCN. All types defined in ASN.1 and in this clause may be referenced even though they do not appear in a type definition in a test suite. All other types used in a test suite shall be declared in the Test Suite Type Definitions, ASP definitions or PDU definitions and referenced by name.

The following TTCN predefined types are considered to be the same as their counterparts in ASN.1:

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

   Values of type INTEGER 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) **BOOLEAN predefined type**: A type consisting of two distinguished values.

   Values of the BOOLEAN type are TRUE and FALSE.

c) **BITSTRING predefined type**: 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 ' and followed by the pair of characters 'B:

   EXAMPLE 8 – '01101'B

d) **HEXSTRING predefined type:** A type whose distinguished values are the ordered sequences of zero, one, or more HEX 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 HEX digits:

   0 1 2 3 4 5 6 7 8 9 A B C D E F

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

   EXAMPLE 9 – 'AB01D'H

e) **OCTETSTRING predefined type**: A type whose distinguished values are the ordered sequences of zero or a positive even number of HEX 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 HEX digits:

0 1 2 3 4 5 6 7 8 9 A B C D E F

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

EXAMPLE 10 – 'FF96'O

f) **OBJECTIDENTIFIER predefined type:** A type whose distinguished values are the set of all object identifiers allocated in accordance with the rules of Recommendation X.680.

g) **R_TYPE predefined type:** A type consisting of the following distinguished values:

pass, fail, inconc and none

These values are predefined identifiers and as such, are case sensitive. This predefined type is for use with verdicts, see 15.17.

h) **CharacterString predefined types**: Types whose distinguished values are zero, one, or more characters from some character set; the CharacterString types listed in Table 2 may be used; they are defined in clause 31/X.680.

Table 2/X.292 – Predefined CharacterString Type

NumericString
PrintableString
TeletexString
T61String
VideotexString
VisibleString
ISO646String
IA5String
GraphicString
GeneralString
BMPString
UniversalString

Values of CharacterString types shall be denoted by an arbitrary number (possibly zero) of characters from the character set referenced by the CharacterString type, preceded and followed by double quote ("); if the CharacterString type includes the character double quote, this character shall be represented by a pair of double quote in the denotation of any value.

SYNTAX DEFINITION:

735 PredefinedType ::= **INTEGER** | **BOOLEAN** | **BITSTRING** | **HEXSTRING** | **OCTETSTRING** | **OBJECTIDENTIFIER** | **R_Type** | CharacterString

736 CharacterString ::= **NumericString** | **PrintableString** | **TeletexString** | **VideotexString** | **VisibleString** | **IA5String** | **GraphicString** | **GeneralString** | **T61String** | **ISO646String**

741 **Number ::= (NonZeroNum {Num}) | 0**

742 **NonZeroNum ::= 1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

743 **Num ::= 0** | **NonZeroNum**

744 **BooleanValue ::= TRUE** | **FALSE**

745 **Bstring ::= "_" {Bin | Wildcard} "_" B**

746 **Bin ::= 0** | **1**

747 **Hstring ::= "_" {Hex | Wildcard} "_" H**

748 **Hex ::= Num | A** | **B** | **C** | **D** | **E** | **F**

749 **Ostring ::= "_" {Oct | Wildcard} "_" O**

750 **Oct ::= Hex Hex**

751 **Cstring ::= """ {Char | Wildcard |** "\"**} """**

752 Char ::= /* REFERENCE – A character defined by the relevant CharacterString type. */

753 Wildcard ::= AnyOne | *AnyOrNone*

754 AnyOne ::= "?"

755 AnyOrNone ::= ""*

11.2.3 Test Suite Type Definitions

11.2.3.1 Introduction

Type definitions to be used as types for data objects and as subtypes for structured ASPs, PDUs, etc., can be introduced using a tabular format and/or ASN.1. Wherever types are referenced within Test Suite Type definitions those references shall not be recursive (neither directly or indirectly).

11.2.3.2 Simple Type Definitions using tables

To define a new Simple Type, the following information shall be provided:

a) a name for the type;

b) the base type,

where the base type shall be a Predefined Type or a Simple Type. The base type is followed by the type restriction that shall take one of the following forms:

1) A list of distinguished values of the base type; these values comprise the new type.

2) A specification of a range of values of type INTEGER; the new type comprises the values including the lower boundary and the upper boundary specified in the range. In order to specify an infinite range, the keyword INFINITY may be used instead of a value indicating that there is no upper boundary or lower boundary.

3) A specification of a particular length or length range of a predefined or test suite string type; the length value(s) shall be interpreted according to Table 5 in 11.18.2; only non-negative INTEGER literals or the keyword INFINITY for the upper bound shall be used.

c) Optionally, a specific encoding identifier followed by any necessary actual parameter list, in order to specify an explicit encoding for the simple type, which overrides the encoding rules and encoding variations applicable to any PDU in which that simple type is used; the encoding identifier, if any, shall identify either one of the Encoding Variations or an Invalid Field Encoding Definition defined in the test suite [e.g. LD(10)]; see 11.16.4.

This information shall be provided in the format shown in Proforma 7, below.

Simple Type Definitions			
Group : *[SimpleTypeGroupReference]*			
Type Name	**Type Definition**	**Type Encoding**	**Comments**
⋮ *SimpleTypeIdentifier* ⋮	⋮ *Type&Restriction* ⋮	⋮ *[PDU_FieldEncodingCall]* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*			

Proforma 7 – Simple Type Definitions

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

75 SimpleTypeGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {SimpleTypeGroupIdentifier "/"}

79 SimpleTypeIdentifier ::= Identifier

80 Type&Restriction ::= Type[Restriction]

515 PDU_FieldEncodingCall ::= EncVariationCall | InvalidFieldEncodingCall

Where a range is used in a type definition either as a value range or as a length range (for strings), it shall be stated with the lower of the two values on the left. An integer range shall be used only with a base type of INTEGER or a type derived from INTEGER. In the latter case, integer range shall be a subrange of the set of values defined by the base type.

Where a value list is used, the values shall be of the base type and shall be a true subset of the values defined by the base type. Where a length restriction is used, the set of values for a type defined by this restriction shall be a true subset of the values defined by the base type.

Values of any two simple types which have the same base type are considered to be type compatible (e.g. for the purposes of performing assignments or parameter passing).

EXAMPLE 11 – Simple Test Suite Type definitions:

Simple Type Definitions		
Type Name	**Type Definition**	**Comments**
Transport_classes String5 SeqNumbers PositiveNumbers String 10 to 20	INTEGER(0, 1, 2, 3, 4) IA5string[5] INTEGER(0,127) INTEGER(1...INFINITY) IA5String[10..20]	Classes that may be used for transport layer connection String of length 5 All numbers from 0 to 127 All positive INTEGER numbers String, min. length 10 characters and max, length 20 characters

11.2.3.3 Structured Type Definitions using tables

Structured Types can be defined in the tabular form to be used for declaring structured objects as subtypes within ASP and PDU definitions and other Structured Types, etc.

The following information shall be supplied for each Structured Type:

a) Its name,

   where appropriate the full name, as given in the relevant protocol Recommendation, shall be used; if an abbreviation is used, then the full name shall follow in parentheses.

b) The Encoding Variations to be used for structures of this type within a PDU.

   In order to specify explicit Encoding Variations for entire structured types, which override the Encoding Variations applicable to any PDU in which this structured type is used, this optional entry shall reference an entry in the relevant Encoding Variations table [e.g. to change from SD to LD(3)]. If this entry is not used, then the applicable Encoding Variations are those applicable to each PDU within which this structured type is used. See 11.16.4.

c) A list of the elements associated with the Structured Type,

   where the following information shall be supplied for each element:

   1) its name,

      where the full name, as given in the appropriate protocol Recommendation, shall be used; if an abbreviation is used, then the full name shall follow in parentheses;

   2) its type and an optional attribute,

      where elements may be of a type of arbitrarily complex structure; there shall be no recursive references (neither directly nor indirectly);

      the optional element length restriction can be used in order to give the minimum and maximum length of an element of a string type (see 11.18);

   3) optionally, a specific encoding identifier followed by any necessary actual parameter list, in order to specify an explicit encoding for the structured type, which overrides the encoding rules and encoding variations applicable to any PDU in which that structured type is used; the encoding identifier, if any, shall identify either one of the Encoding Variations or an Invalid Field Encoding Definition defined in the test suite [e.g. LD(10)]; see 11.16.4.

The elements of Structured Type definitions are considered to be optional, i.e., in instances of these types whole elements may not be present.

This information shall be provided in the format shown in Proforma 8, below.

Structured Type Definition			
Type Name : *StructId&FullId*			
Group : *[StructTypeGroupReference]*			
Encoding Variation : *[EncVariationCall]*			
Comments : *[FreeText]*			
Element Name	**Type Definition**	**Field Encoding**	**Comments**
⋮ *ElemId&FullId* ⋮	⋮ *Type&Attributes* ⋮	⋮ *[PDU_FieldEncodingCall]* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*			

Proforma 8 – Structured Type Definition

SYNTAX DEFINITION:

97 StructId&FullId ::= StructIdentifier [fullIdentifier]

101 StructTypeGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {StructTypeGroupIdentifier "/"}

511 EncVariationCall ::= EncVariationIdentifier [ActualParList]

106 ElemId&FullId ::= ElemIdentifier [FullIdentifier]

396 Type&Attributes ::= (Type [LengthAttribute]) | PDU

515 PDU_FieldEncodingCall ::= EncVariationCall | InvalidFieldEncodingCall

11.2.3.4 Test Suite Type Definitions using ASN.1

Test Suite Types can be specified using ASN.1 This shall be achieved by an ASN.1 definition using the ASN.1 syntax as defined in Recommendation X.680. The following information shall be supplied for each ASN.1type:

a) Its name,

where appropriate the full name, as given in the relevant protocol Recommendation, shall be used; if an abbreviation is used, then the full name shall follow in parentheses.

b) The Encoding Variations to be used for structures of this type within a PDU.

In order to specify explicit Encoding Variations for entire ASN1_Types, which override the Encoding Variations applicable to any PDU in which this ASN1_Type is used, this optional entry shall reference an entry in the relevant Encoding Variations table [e.g. to change from SD to LD(3)]. If this entry is not used, then the applicable Encoding Variations are those applicable to each PDU within which this ASN1_Type is used. See 11.16.4.

c) The ASN.1 type definition,

which shall follow the syntax defined in Recommendations X.680, except that there is the additional option of specifying an Encoding Variation or Invalid Field Encoding associated with either the whole ASN1_Type or any ASN.1 Type within the ASN1_Type. This is done by giving a specific encoding identifier followed by any necessary actual parameter list, in order to specify explicit encodings for individual fields or other subtypes of a PDU, which override the encoding rules and encoding variations applicable to the PDU as a whole; the encoding identifier, if any, shall identify either one of the Encoding Variations or an Invalid Field Encoding Definition defined in the test suite [e.g. LD(10)]; see 11.16.4.

For identifiers within that definition the hyphen symbol (-) shall not be used. The underscore symbol (_) may be used instead. The type identifier in the table header is the name of the first type defined in the table body.

Types referred to from the type definition shall be defined in other ASN.1 Type definition tables, be defined by reference in the ASN.1 type reference table or be defined locally in the same table, following the first type definition. Locally defined types shall not be used in other parts of the test suite.

ASN.1 type definitions used within TTCN shall not use external type references as defined in Recommendations X.680. ASN.1 comments can be used within the table body. The comments column shall not be present in this table.

Comments in ASN.1 start with "--" and end with either the next occurrence of "--" or with "end of line", whichever comes first. This prevents a single ASN.1 comment from spanning several lines. "End of line" is not, however, a defined symbol in TTCN.MP. ATS specifiers are recommended to facilitate the exchange of ATSs in TTCN.MP by always closing ASN.1 comments with "--".

This information shall be provided in Proforma 9, below.

ASN.1 Type Definition	
Type Name	**:** *ASN1_TypeId&FullId*
Group	**:** *[ASN1_TypeGroupReference]*
Encoding Variation	**:** *[EncVariationCall]*
Comments	**:** *[FreeText]*
Type Definition	
⋮ *ASN1_Type&LocalTypes* ⋮	
Detailed Comments:	*[FreeText]*

Proforma 9 – ASN.1 Type Definition

SYNTAX DEFINITION:

115 ASN1_TypeId&FullId ::= ASN1_TypeIdentifier [FullIdentifier]

118 ASN1_TypeGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ASN1_TypeGroupIdentifier "/"}

511 Enc VariationCall ::= EncVariationIdentifier [ActualParList]

121 ASN1_Type&LocalTypes ::= ASN1_Type {ASN1_LocalType}

EXAMPLE 12 – An ASN.1 Test Suite Type Definition:

ASN.1 Type Definition	
Type Name	**:** DATE_type
Comments	**:** To illustrate the structure of ASN.1 type definitions
Type Definition	

SEQUENCE {
day DAY_type,
month MONTH_type,
year YEAR_type
}
-- *local DAY_type*
DAY_type ::= INTEGER {first(), last(31)}
-- *MONTH_type and YEAR_type are defined ASN.1 Type Definitions tables*	

11.2.3.5 ASN.1 Type Definitions by Reference

Types can be specified by a precise reference to an ASN.1 type defined in an OSI Recommendation or by referencing an ASN.1 type defined in an ASN.1 module attached to the test suite. The following information shall be supplied for each type:

a) Its name,

   where this name may be used throughout the entire test suite. This name shall be specified without a FullIdentifier.

b) The type reference,

   which shall follow the identifier rules stated in Recommendations X.680.

c) The module identifier,

   which consists of a module reference that shall follow the identifier rules stated in Recommendations X.680, and an optional ObjectIdentifier; the module shall be unique within the domain of interest.

d) The Encoding Variations to be used for such ASN1_Types within a PDU.

   In order to specify explicit Encoding Variations for entire ASN1_Types, which override the Encoding Variations applicable to any PDU in which this ASN1_Type is used, this optional entry shall reference an entry in the relevant Encoding Variations table [e.g. to change from SD to LD(3)]. If this entry is not used, then the applicable Encoding Variations are those applicable to each PDU within which this ASN1_Type is used. See 11.16.4.

This information shall be provided in Proforma 10, below.

ASN.1 Type Definitions by Reference				
Group : *[ASN1_TypeGroupReference]*				
Type Name	**Type Reference**	**Module Identifier**	**Encoding Variation**	**Comments**
⋮	⋮	⋮	⋮	⋮
ASN1_TypeId&FullId	*TypeReference*	*ASN1_ModuleIdentifier*	*[EncVariationCall]*	*[FreeText]*
⋮	⋮	⋮	⋮	⋮
Detailed Comments: *[FreeText]*				

Proforma 10 – ASN.1 Type Definitions by Reference

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

118 ASN1_TypeGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ASN1_TypeGroupIdentifier "/"}
115 ASN1_TypeId&FullId ::= ASN1_TypeIdentifier [FullIdentifier]
131 TypeReference ::= typereference
133 ASN1_ModuleIdentifier ::= ModuleIdentifier
511 EncVariationCall ::= EncVariationIdentifier [ActualParList]

Since the ASN.1 types imported from ASN.1 modules can contain identifiers, type references and value references that follow the identifier rules in Recommendations X.680, they can contain hyphens. To be able to use the imported definitions in TTCN it is necessary to change the hyphens in imported identifiers to underscore. This is done in the import process.

EXAMPLE 13 – The following type definition in an ASN.1 module:

```
module-1 DEFINITIONS BEGIN
    Type-1 ::= SEQUENCE          {   field1    Sub-Type-1,
                                     field2    BIT STRING {first-bit(0), second-bit(1) } }
END
```

can be imported to TTCN with:

ASN.1 Type Definitions by Reference			
Type Name	**Type Reference**	**Module Identifier**	**Comments**
Type_1 Sub_Type_1	Type-1 Sub-Type-1	module-1 module-1	

The above reference definition of Type-1 is equivalent to the following definition:

ASN.1 Type Definition		
Type Name	**:** Type_1	
Comments	**:**	
Type Definition		
SEQUENCE {	field	Sub_Type_1,
	Field	BIT STRING {first_bit(0), second_bit(1) } }

11.3 TTCN operators and TTCN operations

11.3.1 Introduction

TTCN supports a number of predefined operators, operations and mechanisms that allow the definition of Test Suite Operations. These operators and operations may be used throughout any dynamic behaviour descriptions and constraints.

11.3.2 TTCN operators

11.3.2.1 Introduction

The predefined operators fall into three categories:

a) arithmetic;

b) relational;

c) Boolean.

The precedence of these operators is shown in Table 3. Parentheses may be used to group operands in expressions, a parenthesized expression has the highest precedence for evaluation.

Within any row in Table 3, the listed operators have equal precedence. If more than one operator of equal precedence appear in an expression, the operations are evaluated left to right.

Table 3/X.292 – Precedence of Operators

Highest ↓ Lowest	Unary	( ) + – NOT
	Binary	* / MOD AND + – OR = < > <> >= <=

SYNTAX DEFINITION:

```
723  AddOp ::= "+"|"–"| OR
724  MultiplyOp ::= "*"|"/"| MOD | AND
725  UnaryOp  ::= "+"|"–"| NOT
726  RelOp ::= "="|"<"|">"|"<>"|">="|"<="
```

11.3.2.2 Predefined arithmetic operators

The predefined arithmetic operators are:

"+", "–", "*", "/", **MOD**

They represent the operations of addition, subtraction, multiplication, division and modulo. Operands of these operators shall be of type INTEGER (i.e., TTCN or ASN.1 predefined) or derivations of INTEGER (i.e., subrange). ASN.1 Named Values shall not be used within arithmetic expressions as operands of operations.

The result type of arithmetic operations is INTEGER.

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 value resulting from dividing the first INTEGER by the second (i.e., fractions are discarded).

The result of performing the MOD operation on two INTEGER values gives the remainder of dividing the first INTEGER by the second.

11.3.2.3 Predefined relational operators

The predefined relational operators are:

"=" | "<" | ">" | "<>" | ">=" | "<="

They represent the relations of equality, less than, greater than, not equal to, greater than or equal to and less than or equal to. Operands of equality (=) and not equal to (<>) may be of an arbitrary type. The two operands shall be compatible. All other relational operators shall have operands only of type INTEGER or derivatives of INTEGER. The result type of these operations is BOOLEAN.

In string comparisons BITSTRING, HEXSTRING, OCTETSTRING and all kinds of CharacterStrings may contain the wildcard characters AnyOrNone (*) and AnyOne (?). In this case the comparison is performed according to the pattern matching rules defined in 12.6.2.

11.3.2.4 Predefined Boolean operators

The predefined Boolean operators are:

NOT AND OR

They represent the operations of negation, logical AND and logical OR. Their operands shall be of type BOOLEAN (TTCN or ASN.1 or predefined). The result type of the Boolean operators is BOOLEAN. 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 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.

11.3.3 Predefined operations

11.3.3.1 Introduction

11.3.3.1.1 The predefined operations fall into two categories:

a) conversion;

b) others.

11.3.3.1.2 Predefined operations may be used in every test suite. They do not require an explicit definition using a Test Suite Operation Definition table. When a predefined operation is invoked:

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

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

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

Each of the predefined operations is presented in the following format:

OPERATION_NAME (FORMAL_PARAMETER_LIST) ➔ RESULT_TYPE

11.3.3.2 Predefined conversion operations

11.3.3.2.1 TTCN supports the following predefined operations for type conversions:

a) HEX_TO_INT converts HEXSTRING to INTEGER;

b) BIT_TO_INT converts BITSTRING to INTEGER;

c) INT_TO_HEX converts INTEGER to HEXSTRING;

d) INT_TO_BIT converts INTEGER to BITSTRING.

These operations provide encoding rules within the context of the operations only. It is invalid to assume these encoding rules apply outside the domain of the operations in TTCN.

11.3.3.2.2 HEX_TO_INT(hexvalue:HEXSTRING) ➔ INTEGER

This operation 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 HEX digit is least significant, the leftmost HEX digit is the most significant. The HEX digits 0 .. F represent the decimal values 0 .. 15 respectively.

11.3.3.2.3 BIT_TO_INT(bitvalue:BITSTRING) ➔ INTEGER

This operation 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.

11.3.3.2.4 INT_TO_HEX(intvalue, slength:INTEGER) ➔ HEXSTRING

This operation converts a single INTEGER value to a single HEXSTRING value. The resulting string is *slength* HEX digits long.

For the purposes of this conversion, a HEXSTRING shall be interpreted as a positive base 16 INTEGER value. The rightmost HEX digit is least significant, the leftmost HEX digit is the most significant. The HEX digits 0 .. F represent the decimal values 0 .. 15 respectively.

If the conversion yields a value with fewer HEX digits than specified in the second parameter, then the HEXSTRING shall be padded on the left with zeros.

A test case error shall occur if the *intvalue* is negative or if the resulting HEXSTRING contains more HEX digits than specified in the second parameter.

11.3.3.2.5 INT_TO_BIT(intvalue, slength:INTEGER) ➔ BITSTRING

This operation converts a single INTEGER value to a single BITSTRING value. The resulting string is *slength* 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 second parameter, then the BITSTRING shall be padded on the left with zeros.

A test case error shall occur if the *intvalue* is negative or if the resulting BITSTRING contains more bits than specified in the second parameter.

11.3.3.3 Other predefined operations

TTCN also defines the following predefined operations:

a) IS_PRESENT;

b) IS_CHOSEN;

c) NUMBER_OF_ELEMENTS;

d) LENGTH_OF;

e) SIZE_OF.

11.3.3.3.1 IS_PRESENT(DataObjectReference) ➔ BOOLEAN

As an argument the operation shall take a reference to a field within a data object only if it is defined as being OPTIONAL or if it has a DEFAULT value. The field may be of any type. The result of applying the operation is the BOOLEAN value TRUE if and only if the value of the field is present in the actual instance of the data object. Otherwise the result is FALSE.

The argument of the operation shall have the format as defined in 15.10.2.

EXAMPLE 14 – Use of IS_PRESENT:

```
if received_PDU is of ASN.1 type
    SEQUENCE    {    field_1 INTEGER OPTIONAL,
                     field_2 SEQUENCE OF INTEGER        }
then, the operation call
    IS_PRESENT(received_PDU.field_1)
evaluates to TRUE if field_1 in the actual instance of received_PDU is present.
```

11.3.3.3.2 IS_CHOSEN(DataObjectReference) ➔ BOOLEAN

The operation returns the BOOLEAN value TRUE if and only if the data object reference specifies the variant of the CHOICE type that is actually selected for a given data object. Otherwise the result is FALSE. The operation shall not be applied to data objects or fields of data objects other than those of ASN.1 type CHOICE. The argument of the operation shall have the format as defined in 15.10.2.

EXAMPLE 15 – Use of IS_CHOSEN:

```
if received_PDU is of ASN.1 type
    CHOICE  {        p1 PDU_type1,
                     p2 PDU_type2,
                     p3 PDU_type }
then, the operation call
    IS_CHOSEN(received_PDU.p2)
returns TRUE if the actual instance of received_PDU carries a PDU of the type PDU_type2.
```

11.3.3.3.3 NUMBER_OF_ELEMENTS(Value) ➔ INTEGER

The operation returns the actual number of elements of a value that is of type ASN.1 SEQUENCE OF or SET OF. Its result is fully compatible with that of the equivalent ASN.1 SIZE constraint applied to objects of these types. The operation shall not be applied to values other than of ASN.1 type SEQUENCE OF or SET OF. The argument of the operation shall have the format as defined in 15.10.2.

EXAMPLE 16 – Use of NUMBER_OF_ELEMENTS:

```
if received_PDU is of ASN.1 type
    SEQUENCE    {    field_1 INTEGER OPTIONAL,
                     field_2 SEQUENCE OF INTEGER }
then, the operation call
    NUMBER_OF_ELEMENTS(received_PDU.field_2)
returns the number of elements of the SEQUENCE OF INTEGER within the actual data object received_PDU.
Also, NUMBER_OF_ELEMENTS ({3, 0, 5}) returns 3.
```

11.3.3.3.4 LENGTH_OF(Value) ➔ INTEGER

The operation returns the actual length of a value that is of type BITSTRING, HEXSTRING, OCTETSTRING, or CharacterString or of ASN.1 type BIT STRING or OCTET STRING. The units of length for each string type are defined in Table 5 in 11.18.2.

NOTE – These units of length are compatible with those used in ASN.1 SIZE constraints for objects of ASN.1 types, but not for literal values which in this context in TTCN are considered to be of the corresponding TTCN type. Thus, an hstring such as 'F3'H which could in ASN.1 be of type BIT STRING or OCTET STRING, will be interpreted as the TTCN type HEXSTRING.

The argument of the operation shall have the format as defined in 15.10.2.

The operation shall not be applied to values other than of type BITSTRING, HEXSTRING, OCTETSTRING, or CharacterString, or of ASN.1 type BIT STRING or OCTET STRING.

EXAMPLE 17 – Use of LENGTH_OF:

If S is of type BITSTRING or ASN.1 type BIT STRING and =’010’B, then LENGTH_OF(S) returns 3.
If S is of type HEXSTRING and =’F3’H, then LENGTH_OF(S) returns 2.
If S is of type OCTETSTRING and =’F2’O, then LENGTH_OF(S) returns 1.
If S is of a CharacterString type and ="EXAMPLE", then LENGTH_OF(S) returns 7.
If S is of ASN.1 type BIT STRING and =’F3’H, thenLENGTH_OF(S) returns 8.
If S is of ASN.1 type OCTET STRING and =’F3’H, then LENGTH_OF(S) returns 1.
If S is of ASN.1 type OCTET STRING and =’01010011’B, then LENGTH_OF(S) returns 1.
Also, LENGTH_OF (INT_TO_HEX (26, 4)) returns 4.
LENGTH_OF (’F3’H) returns 2
and, LENGTH_OF ("Length_of Example") returns 17.

11.3.4 Test Suite Operation definitions and descriptions

11.3.4.1 Introduction

Operations specific to a Test Suite may be defined by the ATS specifier. To define a new operation, the following shall be provided:

a) A name for the operation.

b) A list of the input parameters and their types.

This is a list of the formal parameter names and types. Each parameter name shall be followed by a colon and then the name of the parameter type.

When more than one parameter of the same type is used, the parameters may be specified as a parameter sublist. When a parameter sublist is used, the parameter names shall be separated from each other by a comma. The final parameter in the list shall be followed by a colon and then the name of the type of the parameter.

When more than one parameter and type pair (or parameter list and type pair) is used, the pairs shall be separated from each other by semicolons.

Only predefined types and data types as defined in the Test Suite Type definitions, ASP type definitions or PDU type definitions may be used as types for formal parameters. PCO types shall not be used as formal parameter types. All parameters shall be passed by value, meaning that in evaluating a call of a test suite operation, the actual parameters are assigned to the corresponding formal parameters, as if in an assignment statement.

EXAMPLE 18 – Parameter lists

The following are equivalent methods of specifying a parameter list using two INTEGER parameters and one BOOLEAN parameter:

(A:INTEGER; B:INTEGER; C:BOOLEAN)

(A, B:INTEGER; C:BOOLEAN)

c) The type of the result,

which shall follow the rules for the parameter types in b).

d) A definition of the operation,

which shall consist of one of the following:

1) a procedural definition, which when evaluated results in the evaluation of a RETURNVALUE statement to provide the result of the operation, including explanatory comments embedded within the procedural definition at appropriate places as text delimited by "/*" and "*/"; or

2) a description of the operation in text, possibly including a reference to a publicly available specification of the algorithm to be applied when the operation is invoked, plus at least one example showing an invocation and corresponding result; the explanation should begin by stating the operation name, followed by a parenthesized list containing the parameter names of the operation; this provides a "pattern" invocation for the operation.

e) Optionally, further comment describing the operation, provided either in the Comments part of the table header or in the Detailed Comments area of the table.

The use of procedural definitions is recommended in order to provide precision in the definition of the operations, but a textual explanation is allowed as an alternative for backwards compatibility.

In the case of a procedural definition, this information shall be provided in the format shown in Proforma 11, below.

Test Suite Operation Procedural Definition	
Operation Name	**:** *TS_ProcId&ParList*
Group	**:** *[TS_ProcGroupReference]*
Result Type	**:** *Type*
Comments	**:** *[FreeText]*
Definition	
TS_OpProcDef	
Detailed Comments:	*[FreeText]*

Proforma 11 – Test Suite Operation Procedural Definition

SYNTAX DEFINITION:

155 TS_Protocol&ParList ::= TS_ProcIdentifier [FormalParList]

158 TS_ProcGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TS_ProcGroupIdentifier "/"}

734 Type ::= PredefinedType | Reference Type

162 TS_OpProcDef ::= [VarBlock] ProcStatement

In the case of a textual description, this information shall be provided in Proforma 12, below.

Test Suite Operation Description	
Operation Name	**:** *TS_OpId&ParList*
Group	**:** *[TS_OpGroupReference]*
Result Type	**:** *Type*
Comments	**:** *[FreeText]*
Description	
FreeText	
Detailed Comments:	*[FreeText]*

Proforma 12 – Test Suite Operation Description

SYNTAX DEFINITION:

141 TS_OpId&ParList ::= TS_OpIdentifier [FormalParList]

144 TS_OpGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TS_OpGroupIdentifier "/"}

734 Type ::= PredefinedType | ReferenceType

11.3.4.2 Parameters

A Test Suite Operation may be compared to a function in an ordinary programming language. Values shall only be passed into the operation by formal parameters. Each formal parameter shall be declared to be a Predefined Type, a Test Suite Type

Identifier, ASP Type Identifier, PDU Type Identifier, CM Type Identifier or the meta-type **PDU**. Test Suite variables, Test Case variables, Test Suite Constants, Test Suite Parameters and constraints shall not directly be used within the procedural definition of a Test Suite Operation, but if required in the Test Suite operation shall be passed as actual parameters.

There shall be no side-effects, that is, the parameters to the operation shall not be altered as a result of any call of the operation. Predefined operations and other Test Suite Operations may be used within the procedural definition of a Test Suite Operation, without having to be passed as actual parameters.

When a Test Suite Operation is invoked:

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

b) each actual parameter shall evaluate to an element of its corresponding formal parameter type;

c) all variables appearing in the actual parameter list shall be bound; and

d) the actual parameters shall be passed by value.

11.3.4.3 Variables and Identifiers

If a procedural definition is used, it may include the declaration of local variables, placed at the head of the procedural definition, between the keywords **VAR** and **ENDVAR.** These variables may be of any type allowed in TTCN. The scope of these local variables is the procedural definition itself. These declarations declare lists of variable identifiers, each of a given type and each list may either be declared to be **STATIC** or not. Variables, both **STATIC** and those not declared as **STATIC**, may be given an optional initial value.

NOTE – It is recommended always to provide **STATIC** variables with an initial value.

The variables which are not declared to be **STATIC** are initialized every time the operation is invoked, with the specified initial value, if any, and thus they shall not convey a value from one evaluation of the Test Suite Operation to another. Those which are declared to be **STATIC** are initialized with the specified initial value, if any, the first time the operation is invoked within a given test component, or within a given test case if Test Components are not used, and thereafter they retain their values from one invocation to the next within that Test Component or Test Case.

Variables which are not assigned an initial value are considered to be unbound and shall be explicitly bound to a value by an assignment in the operation body before being used in an expression. If an unbound variable is used in an expression, then it is a Test Case error.

Each identifier used in the procedural definition of a Test Suite Operation shall be one of the following:

a) locally declared variable name;

b) a type name, used in a variable declaration;

c) a formal parameter name declared in a formal parameter list of the operation;

d) a Test Suite Operation name.

The scope of formal parameter names and locally declared variable names is the procedural definition of the Test Suite Operation. Thus, the values of all other types of identifier are not directly accessible within the procedural definition of a Test Suite Operation. To access such values, they shall be passed as actual parameters to the Test Suite Operation.

11.3.4.4 Procedure Statements

In a procedural definition, following the declaration of local variables, if any, there shall be a procedure statement of one of the following kinds:

a) a Return statement;

b) an Assignment statement;

c) an If statement;

d) a While loop;

e) a Case statement;

f) a block containing a sequence of procedure statements separated by semicolons and all enclosed by the keywords **BEGIN** and **END**.

Comments may be embedded as text within procedural statements, delimited by "/*" and "*/". Comments shall not be embedded within other comments.

11.3.4.5 ReturnValue statements

Each evaluation of a Test Suite Operation shall end with the evaluation of a ReturnValue statement, consisting of the keyword **RETURNVALUE** followed by an expression. This statement shall return the value of the given expression as the result of the Test Suite Operation. The type of this result shall match the Result Type specified in the header of the Test Suite Operation definition table.

11.3.4.6 Assignment statements

The form of Assignment is the same as in the TTCN behaviour descriptions (see 15.10.4), except that it is not enclosed in parentheses. The DataObjectReference on the left hand side shall begin with a local variable. If the type of the local variable is a structured type, then the DataObjectReference may access a component of that structure (using a record reference, array reference or bit reference, as appropriate, see 15.10.2 and 15.10.3).

11.3.4.7 If statements

There are two forms of If statement:

– **IF** expression **THEN** procedure-statement **ELSE** procedure-statement **ENDIF**;

– **IF** expression **THEN** procedure-statement **ENDIF**.

The expression following the keyword **IF** shall be evaluated first and shall evaluate to a Boolean value. If this evaluates to **TRUE**, then the procedure statement following the keyword **THEN** shall be evaluated. If the expression evaluates to **FALSE**, then the procedure statement following the keyword **ELSE**, if any, is evaluated. The use of the keyword **ENDIF** to end the If statement allows the procedure statements following **THEN** and **ELSE** to be If statements without having to be enclosed in a block.

11.3.4.8 While loop

A While loop takes the form:

– **WHILE** expression **DO** procedure-statement **ENDWHILE**.

The expression following the keyword **WHILE** shall be evaluated first and shall evaluate to a Boolean value. If it evaluates to **TRUE**, then the procedure statement following the keyword **DO** shall be evaluated and then, if no ReturnValue statement has been evaluated, the process shall be repeated starting with the evaluation of the expression again. As soon as the expression evaluates to **FALSE** the evaluation of the While loop is complete.

11.3.4.9 Case statement

A Case statement takes one of the two following forms:

– **CASE** expression **OF**

integer-label_1: procedure-statement_1;
integer-label_2: procedure-statement_2;
...
integer-label_n: procedure-statement_n;

ELSE

procedure-statement

ENDCASE

– **CASE** expression **OF**

integer-label_1: procedure-statement_1;
integer-label_2: procedure-statement_2;
...
integer-label_n: procedure-statement_n;

ENDCASE

The expression following the keyword **CASE** shall be evaluated first and shall evaluate to a positive integer which shall match at most one of the integer labels in the body of the Case statement. The procedure statement following the matched integer label, if any, shall be evaluated and this completes the evaluation of the Case statement. If, however, the result of

evaluating the expression does not match any of the integer labels, then the procedure statement following the keyword **ELSE**, if any, shall be evaluated and this completes the Case statement. If, however, there is no match against an integer label nor an **ELSE** clause, then the result of the Case statement is a Test Case error. Thus, the Case statement is equivalent to a nested sequence of If statements, each testing the expression "(expression) = integer-label_i", possibly followed by an **ELSE** clause at the innermost level of nesting.

11.3.4.10 Use of Test Suite Operations

A Test Suite Operation together with its actual parameter list may be used wherever an expression is allowed.

Each Test Suite Operation should include appropriate error checking. If an error (e.g. division by zero, an invalid parameter, a type mismatch, or evaluation of an unbound variable) is detected during evaluation of a Test Suite Operation, it shall result in a Test Case error.

EXAMPLE 19 – Definition of the operation SUBSTR:

Test Suite Operation Description	
Operation Name	**:** SUBSTR (source:IA5String; start_index, length:INTEGER)
Result Type	**:** IA5String
Description	
SUBSTR(source, start_index, length) is the string of length *len* starting from index *start_index* of the source string *source*. For example: SUBSTR("abcde",3,2) = "cd" SUBSTR("abcde",1,3) = "abc" *SUBSTR(source, start_index, len)* shall only be defined if: *start_index* ≥ 1, *len* ≥ 0, and *start_index* + *len* ≤ *(length of source)* + 1. Any attempt to evaluate SUBSTR applied to arguments on which it is not defined will result in a test case error.	

EXAMPLE 20 – Definition of the operation NUMBER_OF_INVOCATIONS:

Test Suite Operation Procedural Definition	
Operation Name	**:** NUMBER_OF_INVOCATIONS
Result Type	**:** INTEGER
Definition	
VAR STATIC COUNT : INTEGER : 0 ENDVAR BEGIN COUNT := COUNT + 1; **RETURNVALUE** COUNT END	
Detailed Comments: NUMBER_OF_INVOCATIONS() gives an integer value which is equal to the number of times this operation has been invoked in the current test component, or test case if test components are not used.	

11.4 Test Suite Parameter Declarations

The purpose of this part of the ATS is to declare constants derived from the PICS and/or PIXIT which are used to globally parameterize the test suite. These constants are referred to as Test Suite Parameters, and are used as a basis for Test Case selection and parameterization of Test Cases.

The following information relating to each Test Suite Parameter shall be provided:

a) its name;

b) its type,

   where the type shall be a predefined type, an ASN.1 type, a Test Suite Type or a PDU type;

c) its default value, if any,

   which may be used to suggest suitable values for some test suite parameters such as timeout durations;

d) PICS/PIXIT entry reference,

   which is a reference to an individual PICS/PIXIT proforma entry that will clearly identify where the value to be used for this Test Suite Parameter will be found.

This information shall be provided in the format shown in Proforma 13, below.

Test Suite Parameter Declarations				
Group : *[TS_ParGroupReference]*				
Parameter Name	**Type**	**Default Value**	**PICS/PIXIT Ref**	**Comments**
⋮ *TS_ParIdentifier* ⋮	⋮ *Type* ⋮	⋮ *[DefaultValue]* ⋮	⋮ *FreeText* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*				

Proforma 13 – Test Suite Parameter Declarations

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

183 TS_ParGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TS_ParGroupIdentifier "/"}
187 TS_ParIdentifier ::= Identifier
734 Type ::= PredefinedType | ReferenceType

EXAMPLE 21 – Declaration of Test Suite Parameters:

Test Suite Parameter Declarations			
Parameter Name	**Type**	**PIC/PIXIT Ref**	**Comments**
PAR1	INTEGER	PICS question xx	
PAR2	INTEGER	PICS question yy	
PAR3	INTEGER	PICS question zz	

11.5 Test Case Selection Expression Definitions

The purpose of this part of the ATS is to define selection expressions to be used in the Test Case selection process. This part of the ATS shall meet the requirements of Recommendation X.291.

A selection expression is associated with one or more Test Groups and/or Test Cases by placing its identifier in the Test Case Selection Reference column of the Test Suite Structure and/or Test Case Index. An expression may be referenced by more than one Test Group and/or Test Case.

Use of a selection expression shall be taken to mean that the Test Case is to be run if the selection expression evaluates to TRUE.

The following information relating to each Test Case Selection Expression shall be provided:

a) its name;

b) a selection expression,

which shall evaluate to a BOOLEAN value, and which shall use only literal values, Test Suite Parameters, Test Suite Constants and other selection expression identifiers in its terms.

This information shall be provided in the format shown in Proforma 14, below.

Test Case Selection Expression Definitions		
Group : *[SelectionGroupReference]*		
Expression Name	**Selection Expression**	**Comments**
⋮ *SelectExprIdentifier* ⋮	⋮ *SelectionExpression* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*		

Proforma 14 – Test Case Selection Expression Definitions

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

198 SelectExprGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {SelectExprGroupIdentifier "/"}

202 SelectExprIdentifier ::= Identifier

204 SelectionExpression ::= Expression

11.6 Test Suite Constant Declarations

The purpose of this part of the ATS is to declare a set of names for values *not* derived from the PICS or PIXIT that will be constant throughout the Test Suite.

The following information relating to each Test Suite Constant shall be provided:

a) its name;

b) its type,

where the type shall be a predefined type, a simple type or an ASN.1 Type (including PDUs, ASPs and CMs expressed in ASN.1);

c) its value,

where the terms in the value expression shall not contain: Test Suite Variables or Test Case Variables; the value shall evaluate to an element of the type indicated in the type column.

This information shall be provided in the format shown in Proforma 15, below.

Test Suite Constant Declarations			
Group : *[TS_ConstGroupReference]*			
Constant Name	**Type**	**Value**	**Comments**
⋮ *TS_ConstIdentifier* ⋮	⋮ *Type* ⋮	⋮ *DeclarationValue* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*			

Proforma 15 – Test Suite Constant Declarations

Collective comments may be used in this proforma according to Figure 2.

SYNTAX DEFINITION:

212 TS_ConstGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TS_ConstGroupIdentifier "/"}

228 ValueReference ::= valuereference

734 Type ::= PredefinedType | ReferenceType

219 DeclarationValue ::= Expression

EXAMPLE 22 – Declaration of Test Suite Constants:

Test Suite Constant Declarations			
Constant Name	**Type**	**Value**	**Comments**
TS_CONST1	BOOLEAN	TRUE	
TS_CONST2	IA5String	"A string"	

11.7 Test Suite Constant Declarations by Reference

The purpose of this part of the ATS is to declare a set of names for values *not* derived from the PICS or PIXIT that will be constant throughout the Test Suite.

The following information relating to each Test Suite Constant shall be provided:

a) its name;

b) its type,

where the type shall be a predefined type or an ASN.1 type (including PDU, ASP or CM types expressed in ASN.1) imported by an ASN.1 Type Definition By Reference from the ASN.1 module identified by the specified module identifier;

c) its value reference,

where the value shall correspond to an element of the type indicated in the type column;

d) the module identifier,

which consists of a module reference that shall follow the identifier rules stated in Recommendations X.680, and an optional ObjectIdentifier; the module shall be unique within the domain of interest.

This information shall be provided in the format shown in Proforma 16, below.

Test Suite Constant Declarations by Reference				
Group : *[TS_ConstGroupReference]*				
Constant Name	**Type**	**Value Reference**	**Module Identifier**	**Comments**
⋮ *TS_ConstIdentifier* ⋮	⋮ *Type* ⋮	⋮ *ValueReference* ⋮	⋮ *ASN1_ModuleIdentifier* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*				

Proforma 16 – Test Suite Constant Declarations by Reference

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

212 TS_ConstGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TS_ConstGroupIdentifier "/"}
228 ValueReference ::= valuereference
734 Type ::= PredefinedType | ReferenceType
228 ValueReference ::= valuereference
133 ASN1_ModuleIdentifier ::= ModuleIdentifier

11.8 TTCN variables

11.8.1 Test Suite Variable Declarations

A Test Suite may make use of a set of variables which are defined globally for the Test Suite, and retain their values throughout the Test Suite. These variables are referred to as Test Suite Variables.

A Test Suite Variable is used whenever it is necessary to pass information from one Test Case to another. In concurrent TTCN, Test Suite Variables shall only be used by the MTC.

The following information shall be provided for each variable declaration:

a) its name;

b) its type,

   where the type shall be a predefined type, an ASN.1 type, a Test Suite Type or a PDU type;

c) its initial value (if any),

   where the initial value column is used when it is desired to assign an initial value to a Test Suite Variable at its point of declaration; the terms in the value expression shall not contain: Test Suite Variables or Test Case Variables; the value shall evaluate to an element of the type indicated in the Type column. Specifying an initial value is optional.

This information shall be provided in the format shown in Proforma 17, below.

Test Suite Variable Declarations			
Group : *[TS_VarGroupReference]*			
Variable Name	**Type**	**Value**	**Comments**
⋮ *TS_VarIdentifier* ⋮	⋮ *Type* ⋮	⋮ *[DeclarationValue]* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*			

Proforma 17 – Test Suite Variable Declarations

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

235 TS_VarGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TS_VarGroupIdentifier "/"}

239 TS_VarIdentifier ::= Identifier

734 Type ::= PredefinedType | ReferenceType

219 DeclarationValue ::= Expression

Since it is possible that any particular Test Case may be run independently of the others in the Test Suite, it is necessary that the use made of Test Suite Variables does not make assumptions about the ordering of the Test Case execution.

EXAMPLE 23 – Declaration of Test Suite Variables:

Test Suite Variable Declarations			
Variable Name	**Type**	**Value**	**Comments**
state	IA5String	"idle"	Used to indicate the final stable state of the previous Test Case, if any, in order to help determine which preamble to use.

11.8.2 Binding of Test Suite Variables

Initially Test Suite Variables are unbound. They may become bound (or be re-bound) in the following contexts:

a) at the point of declaration if an initial value is specified;

b) when the Test Suite Variable appears on the left-hand side of an assignment statement (see 15.10.4).

Once a Test Suite Variable has been bound to a value, the Test Suite Variable will retain that value until either it is bound to a different value, or execution of the test suite terminates – whichever occurs first.

If an unbound Test Suite Variable is used in the right-hand side of an assignment, then it is a test case error.

11.8.3 Test Case Variable Declarations

A test suite may make use of a set of variables which are declared globally to the Test Suite but whose scope is defined to be local to the Test Case.

In concurrent TTCN, each test component, including the MTC, receives a fresh copy of all Test Case Variables when it is created. These variables are referred to as Test Case Variables.

The following information shall be provided for each variable declaration:

a) its name;

b) its type,

   where the type shall be a predefined type, an ASN.1 type, a Test Suite Type or a PDU type;

c) its initial value (if any),

   where the initial value column is used when it is desired to assign an initial value to a Test Case Variable at its point of declaration; the terms in the value expression shall not contain: Test Suite Variables or Test Case Variables; the value shall evaluate to an element of the type indicated in the type column. Specifying an initial value is optional.

This information shall be provided in the format shown in Proforma 18, below.

Test Case Variable Declarations			
Group : *[TC_VarGroupReference]*			
Variable Name	**Type**	**Value**	**Comments**
⋮ *TC_VarIdentifier* ⋮	⋮ *Type* ⋮	⋮ *[DeclarationValue]* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*			

Proforma 18 – Test Case Variable Declarations

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

248 TC_VarGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TC_VarGroupIdentifier "/"}

252 TC_VarIdentifier ::= Identifier

734 Type ::= PredefinedType | ReferenceType

219 DeclarationValue ::= Expression

NOTE – Caution must be exercised when using Test Case Variables as local variables within a Test Step, in order to avoid usage conflicts with other Test Steps or Test Case Variables. A Test Suite specifier may avoid such problems by adopting a naming convention which will result in all such variables being uniquely named within a Test Suite.

11.8.4 Binding of Test Case Variables

Initially Test Case Variables are unbound. They may become bound (or be re-bound) in the following contexts:

a) at the point of declaration if an initial value is specified;

b) when the Test Case appears on the left-hand side of an assignment statement (see 15.10.4).

Once a Test Case Variable has been bound to a value, the Test Case Variable will retain that value until either it is bound to a different value, or execution of the Test Case terminates – whichever occurs first. At termination of the Test Case, the Test Case Variable becomes re-bound to its initial value, if one is specified, otherwise it becomes unbound.

If an unbound Test Case Variable is used in the right-hand side of an assignment, then it is a test case error.

11.9 PCO Type Declaration

This part of the ATS lists the set of service boundaries where the PCOs (Points of Control and Observation) are located.

The following information shall be provided for each PCO types used in the test suite:

a) its name,

   which is the same name given in the PCO table;

b) its role,

   which shall be declared either as UT or LT in the Role column or by descriptive text in the Comment column; the predefined identifier **UT** indicates that the PCO is an upper tester PCO and **LT** specifies a lower tester PCO; if the Role column is used, then its contents shall be consistent with the role, if any, given in the PCO declaration table.

   NOTE – In a test suite using concurrency, the role of a PCO type may need to be described in terms of the nature of the test component and underlying service provider to be coupled by PCOs of this type.

This information shall be provided in the format shown in Proforma 19, below.

PCO Type Declarations		
Group : *[PCO_GroupReference]*		
PCO Type	**Role**	**Comments**
⋮ *PCO_TypeIdentifier* ⋮	⋮ *[PCO_Role]* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*		

Proforma 19 – PCO Type Declarations

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

271 PCO_GroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {PCO_GroupIdentifier "/"}

263 PCO_TypeIdentifier ::= Identifier

279 PCO_Role ::= **UT** | **LT**

PCO types are defined in the PCO table and therefore the PCO Type table is optional. If a PCO Type is given as an exported object in the Export table, it shall be defined in the PCO Type table.

11.10 PCO Declarations

This part of the ATS lists the set of Points of Control and Observation (PCOs) to be used in the Test Suite and explains where in the testing environment these PCOs exist.

NOTE 1 – The number of PCOs is, where applicable, as defined in Recommendations X.290 and X.291 for the test method(s) identified in the Test Suite Structure table. In TTCN, PCOs may also be used in ways not described in Recommendation X.291, for example to communicate with parts of the Test System or Test Environment not defined in the Test Suite (e.g. to manipulate frequencies or simulate handovers for radio protocol testing).

NOTE 2 – TTCN behaviour statements specified for execution at the UT PCO should not place requirements beyond those specified by Recommendation X.291.

In TTCN the PCO model is based on two First In First Out (FIFO) queues:

– one output queue for sending ASPs and/or PDUs;

– one input queue for receiving ASPs and/or PDUs.

The output queue is assumed to be located within the underlying service-provider or in the case of the UT, within the IUT.

A SEND event at a PCO is successful when the event is passed from the LT to the service-provider, or when the event is passed from the UT to the IUT.

For the purpose of receiving events the tester has an input queue. All incoming events are queued and processed by the tester in the same order they were received, and without loss of any events.

NOTE 3 – The queue model is only an abstract model and is not intended to imply a specific implementation.

The following information shall be provided for each PCO used in the Test Suite:

a) its name,

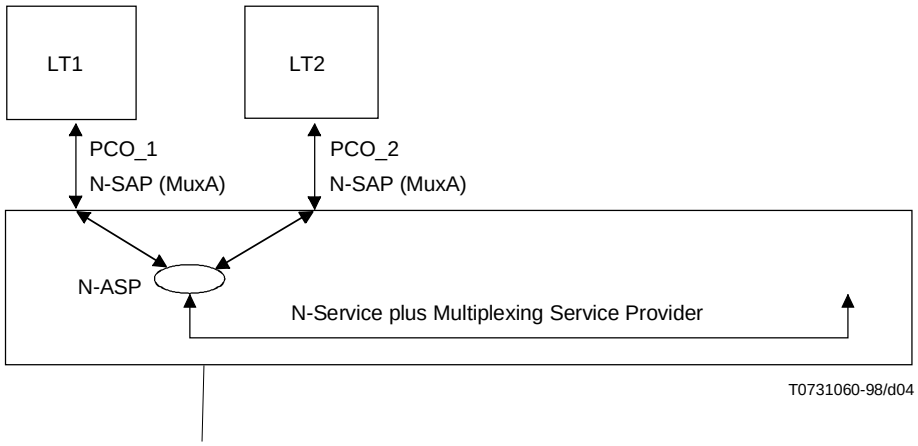
which is used in the behaviour descriptions to specify where particular events occur;

b) its type,

which is used to identify the service boundary where the PCO is located, and which may if necessary be followed by information concerned with multiplexing requirements to be met immediately below this PCO but above the service boundary; if the activity at two or more PCOs is to be multiplexed together by the service provider (e.g. onto a single connection end-point) then, in the PCO declarations for these PCOs, the PCO type shall be followed by the same MuxValue (i.e. a Test Suite Parameter) given in parentheses; the precise meaning of this Test Suite Parameter shall be specified in the relevant PIXIT;

NOTE 4 – See also I.11 for further explanation of MuxValue.

EXAMPLE 24 – Use of MuxValue:



c) its role,

which may be omitted if it is specified in the PCO type declaration table for each of the PCO types used; if the role is not specified in a PCO type declaration table, then it shall be declared either as UT or LT in the Role column or by descriptive text in the Comment column; the predefined identifier **UT** indicates that the PCO is an upper tester PCO and **LT** specifies a lower tester PCO; if the Role column is used, then its contents shall be consistent with the role, if any, given in the PCO type declaration table.

NOTE 5 – In a Test Suite using concurrency, the role of a PCO may need to be described in terms of the nature of the test component and underlying service provider to be coupled by this PCO.

This information shall be provided in the format shown in Proforma 20, below.

PCO Declarations			
Group : *[PCO_GroupReference]*			
PCO Name	**Type**	**Role**	**Comments**
⋮ *PCO_Identifier* ⋮	⋮ *PCO_TypeIdentifier [(MuxValue)]* ⋮	⋮ *[PCO_Role]* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*			

Proforma 20 – PCO Declarations

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

271 PCO_GroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {PCO_GroupIdentifier "/"}

275 PCO_Identifier ::= Identifier

263 PCO_TypeIdentifier ::= Identifier

277 MuxValue ::= TS_ParIdentifier

279 PCO_Role ::= **UT** | **LT**

EXAMPLE 25 – Declaration of PCOs:

PCO Declarations			
PCO Name	**PCO Type**	**Role**	**Comments**
L	TSAP	LT	Transport service access point at the lower tester. Session service access point at the upper tester.
U	SSAP	UT	

Points of control and observation are usually just SAPs, but in general can be any appropriate points at which the test events can be controlled and observed. However, it is possible to define a PCO to correspond to a *set* of SAPs, provided all the Service Access Points (SAPs) comprising that PCO are:

– at the same location (i.e., in the LT or in the UT);

– SAPs of the same service.

When a PCO corresponds to several SAPs, the appropriate address is used to identify the individual SAP. PCOs are normally associated with one service access point of the (N – 1) service-provider or the IUT.

NOTE 6 – A PCO may not be related to a SAP at all. This could be the case when a layer is composed of sublayers (e.g. in the Application layer, or in the lower layers, where a subnetwork point of attachment is not a SAP).

11.11 CP Declarations

CPs are used to facilitate the exchange of CMs between test components. CPs are modelled as two queues, one for each direction of communication. In this respect they are similar to PCOs (see Figure 3). A difference between CPs and PCOs is that CPs connect two test components, while PCOs connect a test component with the external environment, usually either the IUT or a service provider (see Figure 5).

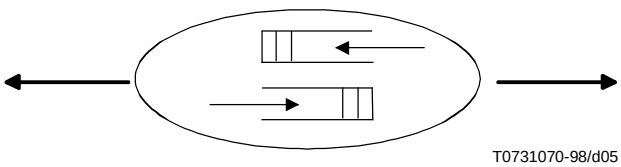


Figure 5/X.292 – Model of a CP

CPs can be realized either by local communication or by communication that spans physical boundaries.

Communication via CPs is asynchronous, that is, communication is achieved by one test component sending a CM to its partner, and its partner receiving the CM when ready. The test component that initiated the CM, however, proceeds with execution immediately after sending the CM. If it is required that the sending test component suspends its activity until the CM has been received, a test suite specifier should use a handshake mechanism. An example of how such a handshake can be specified is shown in Figure 6.

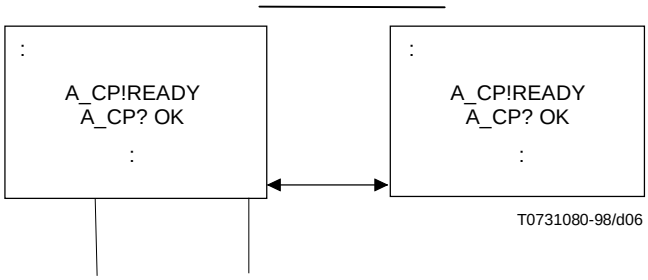


Figure 6/X.292 – Example of a simple handshake

All CPs shall be declared. The name of each CP shall be unique within the Test Suite.

This information shall be provided in the format shown in Proforma 21, below.

CP Declarations	
Group : *[CP_GroupReference]*	
CP Name	**Comments**
⋮ *CP_Identifier* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*	

Proforma 21 – CP Declarations

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

286 CP_GroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {CP_GroupIdentifier"/"}
290 CP_Identifier ::= Identifier

11.12 Timer Declarations

A Test Suite may make use of timers. The following information shall be provided for each timer:

a) the timer name;

b) the optional timer duration,

   where the default duration of the timer shall be an expression which may be omitted if the value cannot be established prior to execution of the Test Suite; the terms in the value expression shall not contain: Test Suite Variables or Test Case Variables; the timer duration shall evaluate to an unsigned positive INTEGER value;

c) the time unit,

   where the time unit shall be one of the following:

   1) **ps** (i.e., picosecond);

   2) **ns** (i.e., nanosecond);

   3) **μs** (i.e., microsecond);

4) **ms** (i.e., millisecond);

5) **s** (i.e., second);

6) **min** (i.e., minute).

Time units are determined by the test suite designer and are fixed at the time of specification. Different timers may use different units within the same Test Suite. If a PICS or PIXIT entry exists, the timer declaration shall specify the same units included in the PICS/PIXIT entry.

This information shall be provided in the format shown in Proforma 22, below.

Timer Declarations			
Group : *[TimerGroupReference]*			
Timer Name	**Duration**	**Unit**	**Comments**
⋮ *TimerIdentifier* ⋮	⋮ *[DeclarationValue]* ⋮	⋮ *TimeUnit* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*			

Proforma 22 – Timer Declarations

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

297 TimerGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TimerGroupIdentifier "/"}

301 TimerIdentifier ::= Identifier

219 DeclarationValue ::= Expression

304 TimerUnit ::= **ps | ns | ms | s | min**

Each Test Component gets a fresh copy of all timers when it starts executing its behaviour.

EXAMPLE 26 – Declaration of timers:

Timer Declarations			
Timer Name	**Duration**	**Unit**	**Comments**
wait	15	s	General purpose wait
no_response	A	min	Used to for IUT to connect or react to connection establish-ment, longer duration than general purpose wait. Gets value from PIXIT.
Delay_time		ms	Duration to be established during execution of the test suite

11.13 Test Components and Configuration Declarations

11.13.1 Test Components

11.13.1.1 Main Test Component

The Main Test Component is intended to fulfil the role of the Lower Tester Control Function (LTCF), as defined in 12.5.2/X.291, Its behaviour is described in the first tree of the Test Case behaviour description table and all trees attached to it. It is responsible for:

a) creating all PTCs required within the current configuration and monitoring their termination;

b) managing CPs that exist between itself and PTCs;

c) computation and assignment of the test verdict using its knowledge of the combined effect of the preliminary results from the PTCs.

In addition a Main Test Component may manage PCO(s).

Only the Main Test Component shall use Test Suite Variables. Test Suite Variables shall not be passed to PTCs in the CREATE construct.

11.13.1.2 Parallel Test Components

Parallel Test Components are intended to fulfil the role of the Lower Testers or Upper Testers. Their behaviour is described in the tree which is referenced in a CREATE statement in the MTC, and all trees attached to it. A PTC assigns preliminary results but does not assign test verdicts.

A PTC shall not:

a) use Test Suite Variables;

b) create other test components.

11.13.1.3 Test Component Declarations

If concurrent TTCN is used, this section of the ATS shall declare all individual test components that are used. These test components are later referenced from the Test Component Configurations declarations which define specific configurations.

The following information shall be provided for each test component:

a) its name,

   which shall be unique throughout the test suite;

b) its role,

   which shall indicate whether the test component is the Main Test Component or a Parallel Test Component, and where at least one test component shall be a Main Test Component, and at least one test component shall be a Parallel Test Component

c) number of PCOs used,

   where zero or more PCOs may be associated with the test component;

d) number of CPs used,

   where zero or more CPs may be associated with the test component.

This information shall be provided in the format shown in Proforma 23, below.

Test Component Declarations				
Group : *[TcompGroupReference]*				
Component Name	**Component Role**	**No. of PCOs**	**No. of CPs**	**Comments**
⋮ *TcompIdentifier* ⋮	⋮ *TCompRole* ⋮	⋮ *Num_PCOs* ⋮	⋮ *Num_CPs* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*				

Proforma 23 – Test Component Declarations

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

311 TcompGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TcompGroupIdentifier "/"}

315 TcompIdentifier ::= Identifier

317 TcompRole ::= MTC | PTC

318 Num_PCOs ::= Number

321 Num_Cos ::= Number

EXAMPLE 27 – Declaration of test components

This Test Component Declarations table can be used in conjunction with the Test Component Configurations CONFIG1 and CONFIG2, illustrated in Figures 3 and 4, and declared in Examples 28 and 29.

Test Component Declarations				
Component Name	**Component Role**	**No. of PCOs**	**No. of CPs**	**Comments**
MTC1	MTC	0	3	Used in Config 1
MTC2	MTC	1	2	Used in Config 2 with a PCO
TC1	PTC	1	2	Used in Config 1
TC2	PTC	1	3	Used in Config 1 and Config 2
TC3	PTC	1	2	Used in Config 1
TC4	PTC	0	3	Used in Config 2
TC5	PTC	1	0	Used in Config 2 without a CP

11.13.2 Test Component Configuration Declarations

Test components are used to build a logical architecture, or configuration, that facilitates concurrent execution of TTCN dynamic behaviour trees. Each Test Component configuration that is used in an Abstract Test Case using concurrency shall be declared.

The following information shall be provided for each Test Component Configuration:

a) its name,

which shall be unique within the Test Suite, and shall be referenced from a Test Case dynamic behaviour table header;

b) a list of the test components belonging to the test configuration,

where the following information shall be provided for each test component:

1) its name,

which shall have been declared as a test component name. Exactly one of the test components in the configuration shall be declared as an MTC;

2) PCOs used,

where a list of zero or more declared PCOs is associated with each test component. The number of PCOs in the list shall be the same as the number of PCOs declared in the relevant Test Components Declaration. No PCO shall be used more than once in a single configuration (i.e. test components in one configuration shall not share PCOs);

3) CPs used,

where a list of zero or more declared CPs is associated with each test component. The number of CPs in the list for a PTC shall be the same as the number of CPs declared in the relevant Test Components Declaration. The number of CPs in the list of an MTC shall not exceed the number of CPs declared. No CP name shall appear more than once in each CP list. Each CP name in the list for one test component shall appear in the list for exactly one other test component in the configuration. In other words, each CP name used in the configuration will appear exactly twice in the configuration table. These CP pairs are used to specify the connectivity of test components in the configuration.

This information shall be provided in the format shown in Proforma 24, below.

Test Component Configuration Declaration			
Configuration Name : *TCompConfigIdentifier*			
Group : *[TCompConfigGroupReference]*			
Comments : *[FreeText]*			
Components Used	**PCOs Used**	**CPs Used**	**Comments**
⋮ *TcompIdentifier* ⋮	⋮ *[PCO_List]* ⋮	⋮ *[CP_List]* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*			

Proforma 24 – Test Component Configuration Declaration

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

329 TcompConfigIdentifier ::= Identifier
330 TcompConfigGroupreference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TcompConfigGroupIdentifier "/"}
315 TcompIdentifier ::= Identifier
337 PCO_List ::= PCO_Identifier {Comma PCO_Identifier}
339 CP_List ::= CP_Identifier {Comma CP_Identifier}

EXAMPLE 28 – Test Component Configuration declaration corresponding to Figure 3:

Test Component Declarations		
Configuration Name : CONFIG_1		
Components Used	**PCOs Used**	**CPs Used**
MTC1		MCP1, MCP2, MCP3
TC1	PCO_A	MCP1, CP1
TC2	PCO_B	MCP2, CP1, CP2
TC3	PCO_C	MCP3, CP2

EXAMPLE 29 – Test Component Configuration declaration corresponding to Figure 4:

Test Component Configuration Declaration		
Configuration Name : CONFIG_2		
Components Used	**PCOs Used**	**CPs Used**
MTC2	PCO_D	MCP2, MCP3
TC2	PCO_B	MCP2, CP1, CP2
TC4		MCP3, CP1, CP2
TC5	PCO_E	

11.14 ASP Type Definitions

11.14.1 Introduction

The purpose of this part of the abstract TTCN test suite is to declare the types of ASPs that may be sent or received at the declared PCOs. ASP type definitions may include ASN.1 type definitions, if appropriate.

11.14.2 ASP Type Definitions using tables

The following information shall be supplied for each ASP:

a) its name,

   where the full name, as given in the appropriate protocol Recommendation, shall be used; if an abbreviation is used, then the full name shall follow in parentheses;

b) the PCO type associated with the ASP,

   where the PCO type shall be one of the PCO types used in the PCO declaration proforma. If only a single PCO is defined within a test suite, specifying the PCO type in an ASP type definition is optional;

c) a list of the parameters associated with the ASP,

   where the following information shall be supplied for each parameter:

   1) its name,

      where either:

      – the full name, as given in the appropriate protocol Recommendation, shall be used; if an abbreviation is used, then the full name shall follow in parentheses; or

      – the macro symbol (<-) indicating that the entry in the type column identifies a set of parameters that is to be inserted directly in the list of ASP parameters; the macro symbol shall be used only with Structured Types defined in the Structured Types definitions;

   2) its type and an optional attribute,

      where parameters may be of a type of arbitrarily complex structure, including being specified as a Test Suite Type (either predefined, Simple Type, Structured Type or ASN.1 type); if a parameter is to be structured as a PDU, then its type may be stated either:

      – as a PDU identifier to indicate that in the constraint for the ASP this parameter may be chained to a PDU constraint of a specific PDU type; or

      – as **PDU** to indicate that in the constraint for the ASP, this parameter may be chained to a PDU constraint of any PDU type; and where the optional attribute is Length;

        in which case the specification may restrict the parameter to a particular length or a range according to 11.18. The length values shall be interpreted according to Table 5 in 11.18. The boundaries shall be specified in terms of non-negative INTEGER literals, Test Suite Parameters, Test Suite Constants or the keyword INFINITY.

The length specifications defined for the ASP parameter type in the Test Suite Type definitions shall not conflict with the length specifications in the ASP type definition, i.e., the set of strings defined by a length restriction in an ASP definition shall be a true subset of the set of strings defined by the Test Suite Type definition.

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.

NOTE – It is usually unnecessary to restrict the length of ASP parameters, but in some cases this may be necessary in order to effectively restrict the length of a corresponding PDU field in an underlying protocol.

The parameters of ASP type definitions are considered to be optional, i.e., in instances of these types whole parameters may not be present.

This information shall be provided in the format shown in Proforma 25, below.

ASP Type Definition		
ASP Name : *ASP_Id&FullId*		
Group : *[ASP_GroupReference]*		
PCO Type : *[PCO_TypeIdentifier]*		
Comments : *[FreeText]*		
Parameter Name	**Parameter Type**	**Comments**
⋮ *ASP_ParIdOrMacro* ⋮	⋮ *Type&Attributes* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*		

Proforma 25 – ASP Type Definition

The Parameter Name and Parameter Type columns shall either be both present or both omitted.

SYNTAX DEFINITION:

348 ASP_Id&FullId ::= ASP_Identifier [FullIdentifier]

351 ASP_Groupreference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ASP_GroupIdentifier "/"}

263 PCO_TypeIdentifier ::= Identifier

357 ASP_ParIdOrMacro ::= ASP_ParId&FullId | MacroSymbol

396 Type&Attributes ::= (Type[LengthAttribute]) |**PDU**

EXAMPLE 30 – T_CONNECTrequest Abstract Service Primitive

The figure below shows an example from the Transport Service (see Recommendation X.214). This could be part of the set of ASPs used to describe the behaviour of an abstract UT in a DS test suite for the Class 0 Transport. CDA,CGA and QOS are Test Suite Types (see Recommendation X.224).

ASP Type Definition		
ASP Name : CONreq (T_CONNECTrequest) **PCO Type** : *TSAP* **Comments** :		
Parameter Used	**Parameter Type**	**Comments**
Cda (Called Address)	CDA	… of upper tester
Cga (Calling Address)	CGA	… of lower tester
QoS (Quality of Service)	QOS	should ensure class 0 is used
Detailed Comments: ASP to be sent at Transport Service access point		

11.14.3 Use of Structured Types within ASP Type Definitions

There are two possible relationships between a Structured Type and ASP definitions which refer to it, as follows:

a) if a parameter name is given in the definition, then the Structured Type referenced is a substructure. This allows definition of ASPs containing a multi-level substructure of parameters;

b) if the macro symbol (<-) is used instead of a parameter name, then this is equivalent to a macro expansion; the entry in the ASP type definition expands directly to a list of parameters without introducing an additional level of substructure.

The macro symbol shall not be used on the same line as references to types defined in ASN.1 or Simple Types, i.e., only Structured Types defined in tabular form can be expanded into other Structured Types as macro expansions.

11.14.4 ASP Type Definitions using ASN.1

Where more appropriate, ASPs can be specified in ASN.1. This shall be achieved by an ASN.1 definition using the ASN.1 syntax as defined in Recommendations X.680. The following information shall be supplied for each ASN.1 ASP:

a) its name,

   where the full name, as given in the appropriate protocol Recommendation, shall be used; if an abbreviation is used, then the full name shall follow in parentheses;

b) the PCO type associated with the ASP,

   where the PCO type shall be one of the PCO types used in the PCO declaration proforma. If only a single PCO is defined within a test suite, specifying the PCO type in an ASP type definition is optional;

c) the ASN.1 ASP type definition,

   which shall follow the syntax defined in Recommendations X.680. For identifiers within that definition the hyphen symbol (-) shall not be used. The underscore symbol (_) may be used instead. The ASP identifier in the table header is the name of the first type defined in the table body.

   Types referred to from the ASP definition shall be defined in other ASN.1 type definition tables, be defined by reference in the ASN.1 type reference table or be defined locally in the same table, following the first type definition. Locally defined types shall not be used in other parts of the test suite.

   ASN.1 comments can be used within the table body. The comments column shall not be present in this table.

Comments in ASN.1 start with "--" and end with either the next occurrence of "--" or with "end of line", whichever comes first. This prevents a single ASN.1 comment from spanning several lines. "End of line" is not, however, a defined symbol in TTCN.MP. ATS specifiers are recommended to facilitate the exchange of ATSs in TTCN.MP by always closing ASN.1 comments with "--".

This information shall be provided in Proforma 26, below.

ASN.1 ASP Type Definition	
ASP Name	**:** *ASP_Id&FullId*
Group	**:** *[ASN1ASP_GroupReference]*
PCO Type	**:** *[PCO_TypeIdentifier]*
Comments	**:** *[FreeText]*
Type Definition	
ASN1_Type&LocalTypes	
Detailed Comments: *[FreeText]*	

Proforma 26 – ASN.1 ASP Type Definition

SYNTAX DEFINITION:

348 ASP_Id&FullId ::= ASP_Identifier [FullIdentifier]
367 ASN1_ASP_GroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ASN1_ASP_GroupIdentifier "/"}
263 PCO_TypeIdentifier ::= Identifier
121 ASN1_Type&Localtypes ::= ASN1_Type {ASN1_LocalType}

11.14.5 ASN.1 ASP Type Definitions by Reference

ASPs can be specified by a precise reference to an ASN.1 ASP defined in an OSI Recommendation or by referencing an ASN.1 type defined in an ASN.1 module attached to the test suite. The following information shall be supplied for each ASP:

a) its name,

   where this name may be used throughout the entire test suite;

b) the PCO type associated with the ASP;

   where the PCO type shall be one of the PCO types used in the PCO declaration proforma. If only a single PCO is defined within a test suite, specifying the PCO type in an ASP type definition is optional;

c) the type reference,

   which shall follow the identifier rules stated in Recommendations X.680;

d) the module identifier,

   which consists of a module reference that shall follow the identifier rules stated in Recommendations X.680 and an optional ObjectIdentifier.

This information shall be provided in Proforma 27, below.

ASN.1 ASP Type Definitions by Reference				
Group **:** *[ASN1ASP_GroupReference]*				
ASP Name	**PCO Type**	**Type Reference**	**Module Identifier**	**Comments**
⋮ *ASP_Id&FullId* ⋮	⋮ *[PCO_TypeIdentifier]* ⋮	⋮ *TypeReference* ⋮	⋮ *ModuleIdentifier* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*				

Proforma 27 – ASN.1 ASP Type Definitions by Reference

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

367 ASN1_ASP_GroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ASN1_ASP_GroupIdentifier "/"}

348 ASP_Id&FullId ::= ASP_Identifier [FullIdentifier]

263 PCO_TypeIdentifier ::= Identifier

131 TypeReference ::= typereference

133 ASN1_ModuleIdentifier ::= ModuleIdentifier

ASN.1 identifiers type references and value references may contain hyphens. In order to be able to use imported definitions in TTCN it is necessary to change the hyphens to underscore (see A.4.2.1).

11.15 PDU Type Definitions

11.15.1 Introduction

The purpose of this part of the abstract TTCN test suite is to declare the types of the PDUs that may be sent or received either directly or embedded in ASPs at the declared PCOs. PDU type definitions may include ASN.1 type definitions, if appropriate. PDU definitions define the set of PDUs exchanged with the IUT which are syntactically valid with respect to the ATS but not necessarily valid with respect to the protocol specification.

It is required to declare all fields of the PDUs that are defined in the relevant protocol Recommendation, either explicitly or implicitly by referring to encoding rules (ASN.1 encoding rules, if applicable).

The encoding of PDU fields shall follow that as defined in the relevant protocol specification unless encoding information is included in the test suite.

11.15.2 PDU Type Definitions using tables

The definition of PDUs is similar to that of ASPs. The following information shall be supplied for each PDU:

a) Its name,

   where the full name, as given in the appropriate protocol Recommendation, shall be used; if an abbreviation is used, then the full name shall follow in parentheses.

b) The PCO type associated with the PDU,

   where the PCO type shall be one of the PCO types used in the PCO declarations; if a PDU is sent or received only embedded in ASPs within the whole test suite, specifying the PCO type is optional; if only a single PCO is defined within a test suite, specifying the PCO type in a PDU type definition is optional.

c) The encoding rules to be used for PDUs of this type.

   In order to specify explicit encodings for entire PDUs, which override the default global encoding rules for the test suite as a whole, this optional entry shall reference an entry in the relevant Encoding Definitions table (e.g. to change from BER to DER). If this entry is not used, then the default global encoding rules apply. See 11.16.4.

d) The Encoding Variations to be used for PDUs of this type.

   In order to specify explicit Encoding Variations for entire PDUs, which override the default global Encoding Variations for the test suite as a whole, this optional entry shall reference an entry in the relevant Encoding Variations table [e.g. to change from SD to LD(3)]. If this entry is not used, then the default global Encoding Variations apply. See 11.16.4.

e) A list of the fields associated with the PDU,

   where the following information shall be supplied for each field:

   1) its name,

where either:

– the full name, as given in the appropriate protocol Recommendation, shall be used; if an abbreviation is used, then the full name shall follow in parentheses; or

– the macro symbol (<-) indicating that the entry in the type column identifies a set of fields that is to be inserted directly in the list of PDU fields; the macro symbol shall be used only with Structured Types defined in the Structured Type definitions;

2) its type and an optional attribute;

where fields may be of a type of arbitrarily complex structure, including being specified as a Test Suite Type (either predefined, Simple Type, Structured Type or ASN.1 type); if a field is to be structured as a PDU, then its type may be stated either:

– as a PDU identifier to indicate that in the constraint for the PDU this field may be chained to a PDU constraint of a specific PDU type; or

– as **PDU** to indicate that in the constraint for the PDU this field may be chained to a PDU constraint of any PDU type;

and where the optional attribute is Length,

in which case the specification may restrict the field to a particular length or a range according to 11.18. The length values shall be interpreted according to Table 5 in 11.18. The boundaries shall be specified in terms of non-negative INTEGER literals, Test Suite Parameters, Test Suite Constants or the keyword INFINITY.

The length specifications defined for the PDU field type in the Test Suite Type definitions shall not conflict with the length specifications in the PDU type definition, i.e., the set of strings defined by a length restriction in a PDU definition shall be a true subset of the set of strings defined by the Test Suite Type definition.

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.

3) Optionally, a specific encoding identifier followed by any necessary actual parameter list, in order to specify explicit encodings for individual fields of a PDU, which override the encoding rules and encoding variations applicable to the PDU as a whole; the encoding identifier, if any, shall identify either one of the Encoding Variations or an Invalid Field Encoding Definition defined in the test suite [e.g. LD(10)]; see 11.16.4.

The fields of PDU type definitions are considered to be optional, i.e., in instances of these types whole fields may not be present.

This information shall be provided in the format shown in Proforma 28, below.

PDU Type Definition			
PDU Name : *PDU_Id&FullId*			
Group : *[PDU_GroupReference]*			
PCO Type : *[PCO_TypeIdentifier]*			
Encoding Rule Name : *[EncodingRuleIdentifier]*			
Encoding Variation : *[EncVariationCall]*			
Comments : *[FreeText]*			
Field Name	**Field Type**	**Field Encoding**	**Comments**
⋮ *PDU_FieldIdOrMacro* ⋮	⋮ *Type&Attributes* ⋮	⋮ *[PDU_FieldEncodingCall]* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*			

Proforma 28 – PDU Type Definition

The Field Name and Field Type columns shall either be both present or both omitted.

SYNTAX DEFINITION:

382 PDU_Id&FullId ::= PDU_Identifier [FullIdentifier]
385 PDU_GroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {PDU_GroupIdentifier "/"}
263 PCO_TypeIdentifier ::= Identifier
452 EncodingRuleIdentifier ::= Identifier
511 EncVariationCall ::= EncVariationIdentifier [ActualParList]
391 PDU_FieldIdORMacro ::= PDU_FieldId&FullId | MacroSymbol
396 Type&Attributes ::= (Type [LengthAttribute]) | PDU
515 PDU_FieldEncodingCall ::= EncVariationCall | InvalidFieldEncodingCall

EXAMPLE 31 – A typical PDU Type Definition:

PDU Type Definition		
PDU Name : INTC (interrupt Confirm) **PCO Type** : NSAP		
Field Name	**field Type**	**Comments**
GFI	BITSTRING	General Format Identifier
LCGN	BITSTRING	Logical Channel Group Number
LCN	BITSTRING	Local Channel Identifier
PTI	OCTETSTRING	Packet Type Identifier
EXTRA	OCTETSTRING	To create long INTC packets

11.15.3 Use of Structured Types within PDU definitions

There are two possible relationships between a Structured Type and PDU definitions which refer to it, as follows:

a) if a field name is given in the definition, then the Structured Type referenced is a substructure. This allows definition of PDUs containing a multi-level substructure of fields;

b) if the macro symbol (<-) is used instead of a field name, then this is equivalent to a macro expansion; the entry in the PDU type definition expands directly to a list of fields without introducing an additional level of substructure.

The macro symbol shall not be used on the same line as references to types defined in ASN.1 or Simple Types i.e., only Structured Types defined in tabular form can be expanded into other Structured Types as macro expansions.

11.15.4 PDU Type Definitions using ASN.1

Where more appropriate, PDUs can be specified in ASN.1. This shall be achieved by an ASN.1 definition using the ASN.1 syntax as defined in Recommendations X.680. The following information shall be supplied for each ASN.1 PDU:

a) Its name,

   where the full name, as given in the appropriate protocol Recommendation, shall be used; if an abbreviation is used, then the full name shall follow in parentheses.

b) The PCO type associated with the PDU,

   where the PCO type shall be one of the PCO types used in the PCO declarations; if a PDU is always sent or received embedded in ASPs, then specification of the PCO type in the PDU type definition is optional; if only a single PCO is defined within a test suite, then specification of the PCO type in the PDU type definition is optional.

c) The encoding rules to be used for PDUs of this type.

   In order to specify explicit encodings for entire PDUs, which override the default global encoding rules for the test suite as a whole, this optional entry shall reference an entry in the relevant Encoding Definitions table (e.g. to change from BER to DER). If this entry is not used, then the default global encoding rules apply. See 11.16.4.

d) The Encoding Variations to be used for PDUs of this type.

In order to specify explicit Encoding Variations for entire PDUs, which override the default global Encoding Variations for the test suite as a whole, this optional entry shall reference an entry in the relevant Encoding Variations table [e.g. to change from SD to LD(3)]. If this entry is not used, then the default global Encoding Variations apply. See 11.16.4.

e) The ASN.1 PDU type definition,

which shall follow the syntax defined in Recommendations X.680, except that there is the additional option of specifying an Encoding Variation or Invalid Field Encoding associated with either the whole ASN1_Type or any ASN.1 Type within the ASN1_Type. This is done by giving a specific encoding identifier followed by any necessary actual parameter list, in order to specify explicit encodings for individual fields or other subtypes of a PDU, which override the encoding rules and encoding variations applicable to the PDU as a whole; the encoding identifier, if any, shall identify either one of the Encoding Variations or an Invalid Field Encoding Definition defined in the test suite [e.g. LD(10)]; see 11.16.4.

For identifiers within that definition the hyphen symbol (-) shall not be used. The underscore symbol (_) may be used instead. The PDU identifier in the table header is the name of the first type defined in the table body.

Types referred to from the PDU definition shall be defined in other ASN.1 type definition tables, be defined by reference in the ASN.1 type reference table or be defined locally in the same table, following the first type definition. Locally defined types shall not be used in other parts of the test suite.

ASN.1 comments may be used within the table body. The comments column shall not be present in this table.

Comments in ASN.1 start with "--" and end with either the next occurrence of "--" or with "end of line", whichever comes first. This prevents a single ASN.1 comment from spanning several lines. "End of line" is not, however, a defined symbol in TTCN.MP. ATS specifiers are recommended to facilitate the exchange of ATSs in TTCN.MP by always closing ASN.1 comments with "--".

This information shall be provided in Proforma 29, below.

ASN.1 PDU Type Definition	
PDU Name	**:** *PDU_Id&FullId*
Group	**:** *[ASN1_PDU_GroupReference]*
PCO Type	**:** *[PCO_TypeIdentifier]*
Encoding Rule Name	**:** *[EncodingRuleIdentifier]*
Encoding Variation	**:** *[EncVariationCall]*
Comments	**:** *[FreeText]*
Type Definition	
ASN1_Type&LocalTypes	
Detailed Comments: *[FreeText]*	

Proforma 29 – ASN.1 PDU Type Definition

SYNTAX DEFINITION:

382 PDU_Id&FullId ::= PDU_Identifier [FullIdentifier]

409 ASN1_PDU_Groupreference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ASN1_PDU_GroupIdentifier"/"}

263 PCO_TypeIdentifier ::= Identifier

452 EncodingRuleIdentifier ::= Identifier

511 EncVariationCall ::= EncVariationIdentifier [ActualParList]

121 ASN1_Type&LocalTypes ::= ASN1_Type {ANS1_LocalType}

EXAMPLE 32 – An FTAM ASN.1 Definition:

ASN.1 PDU Type Definition	
PDU Name	**:** *F_INIT (F_INITIALIZE_response)*
PCO Type	**:**
Comments	**:**
Type Definition	

SEQUENCE {
state_result State_result DEFAULT success,
action_result Action_Result multiple success,
protocol_id Protocol_Version,
}	

11.15.5 ASN.1 PDU Type Definitions by Reference

PDUs can be specified by a precise reference to an ASN.1 PDU defined in an OSI Recommendation or by referencing an ASN.1 type defined in an ASN.1 module attached to the test suite. ASN.1 identifiers, type references and value references may contain hyphens. In order to be able to use imported definitions in TTCN it is necessary to change the hyphens to underscore (see A.4.2.1).

The following information shall be supplied for each PDU:

a) Its name,

   where this name may be used throughout the entire test suite.

b) The PCO type associated with the PDU,

   where the PCO type shall be one of the PCO types used in the PCO declarations; if a PDU is sent or received only embedded in ASPs within the whole test suite, specifying the PCO type is optional; if only a single PCO is defined within a test suite, specifying the PCO type in a PDU type definition is optional.

c) The type reference,

   which shall follow the identifier rules stated in Recommendations X.680.

d) The module identifier,

   which consists of a module reference that shall follow the identifier rules stated in Recommendations X.680 and an optional ObjectIdentifier.

e) The encoding rules to be used for PDUs of this type.

   In order to specify explicit encodings for entire PDUs, which override the default global encoding rules for the test suite as a whole, this optional entry shall reference an entry in the relevant Encoding Definitions table (e.g. to change from BER to DER). If this entry is not used, then the default global encoding rules apply. See 11.16.4.

f) The Encoding Variations to be used for PDUs of this type.

   In order to specify explicit Encoding Variations for entire PDUs, which override the default global Encoding Variations for the test suite as a whole, this optional entry shall reference an entry in the relevant Encoding Variations table [e.g. to change from SD to LD(3)]. If this entry is not used, then the default global Encoding Variations apply. See 11.16.4.

This information shall be provided in Proforma 30, below.

ASN.1 PDU Type Definitions by Reference						
Group : *[ASN1PDU_GroupReference]*						
PDU Name	**PCO Type**	**Type Reference**	**Module Identifier**	**Enc Rule**	**Enc Variation**	**Comments**
⋮ *PDU_Id&FullId* ⋮	⋮ *[PCO_TypeIdentifier]* ⋮	⋮ *TypeReference* ⋮	⋮ *ModuleIdentifier* ⋮	⋮ *[EncodingRuleIdentifier]* ⋮	⋮ *[EncVariationCall]* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*						

Proforma 30 – ASN.1 PDU Type Definitions by Reference

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

409 ASN1_PDU_GroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ASN1_GroupIdentifier "/"}

382 PDU_Id&FullId ::= PDU_Identifier {FullIdentifier}

263 PCO_TypeIdentifier ::= Identifier

131 TypeReference ::= typereference

133 ASN1_ModuleIdentifier ::= ModuleIdentifier

452 EncodingruleIdentifier ::= ModuleIdentifier

511 EncVariationCall ::= EncVariationIdentifier {ActualParList}

11.16 Test Suite Encoding Information

11.16.1 Encoding Definitions

To facilitate specification and testing of the encoding rules of an OSI protocol, if there is any allowed flexibility in the encoding rules applicable to the protocol, then an encoding definition should be provided. If an encoding definition is provided, a reference shall be given in the ATS to the specification in which the encoding rules are specified. The reference may be to the protocol specification itself, or to a separate encoding rules specification. If such a reference cannot be provided, i.e. the encoding rules of the protocol are not standardized, then the encoding rules shall not be tested.

The following information shall be provided for each set of encoding rules relevant to the protocol:

a) the Encoding Rule Name, which is a unique identifier to be used throughout the test suite to refer to an encoding definition;

b) the reference to the relevant Recommendation which defines the encoding rules;

c) a Default Expression, identifying the encoding rules to be used as the default; this Default Expression shall evaluate to a Boolean value and shall use only Literal Values, Test Suite Parameters, and Test Suite Constants in its terms;

d) optionally, further comment, provided in the Comments column, or in the Detailed Comments area of the table.

If more than one set of encoding rules may be used for a protocol, the names of the encoding rules shall be listed in the Encoding Rule Name column of the Encoding Definitions table. The Encoding Rule Name associated with the Default Expression which evaluates to TRUE shall be chosen as the default set for the test suite. If more than one Default Expression or no Default Expression in the Encoding Definitions table evaluates to TRUE, it shall be a test case error. If no Default Expression is specified, it is equivalent to the value FALSE being specified.

The information shall be provided in Proforma 31, below.

Encoding Definitions			
Group : *[EncodingGroupReference]*			
Encoding Rule Name	**Reference**	**Default**	**Comments**
⋮ *EncodingRuleIdentifier* ⋮	⋮ *EncodingReference* ⋮	⋮ *[DefaultExpression]* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*			

Proforma 31 – Encoding Definitions

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

448 EncodingGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {EncodingGroupIdentifier "/"}
452 EncodingRuleIdentifier ::= Identifier
454 EncodingReference ::= BoundedFreeText
456 DefaultExpression ::= Expression

The encoding rules specified in this proforma apply to PDUs only.

EXAMPLE 33 – Encoding Definitions:

Encoding Definitions			
Encoding Rule Name	**Reference**	**Default**	**Comments**
BER	Recommendations X.690	TRUE	Basic Encoding Rules
PER	Recommendations X.690		Packed Encoding Rules
DER	Recommendations X.690		Distinguished Encoding Rules
Detailed Comments: *[FreeText]*			

11.16.2 Encoding Variations

Admissible variations of each encoding definition that may be used in the test suite may be provided.

To define such Encoding Variations, the following information shall be provided:

a) an Encoding Rule Name, which is the name of the encoding rules identified in the Encoding Definition table to which this variation applies;

b) an optional Type List, listing the types to which this Encoding Variation may be applied; an empty list means that the Encoding Variations may be applied to any PDU field. The types may be any PDU type or any type may occur within a PDU;

c) a list of Encoding Variations,

   where the following information shall be supplied for each Encoding Variation:

   1) the Encoding Variation name, which is a unique identifier referring to an allowed encoding definition for a specific type, as contained in the relevant encoding rules specification;

2) a Reference, which is used to identify the section in the encoding rules specification which describes this set of Encoding Variations;

3) a Default Expression, identifying the Encoding Variation to be used as the default; this Default Expression shall evaluate to a Boolean value and shall use only Literal Values, Test Suite Parameters, and Test Suite Constants in its terms;

d) optionally, further comment, provided in the Comments part of the table header, the Comments column, or in the Detailed Comments area of the table.

The Encoding Variation associated with the expression which evaluates to TRUE shall be chosen as the default Encoding Variation for the given list of types, if any, or otherwise for all types within the test suite. If more than one Default Expression in the Encoding Variations table evaluates to TRUE, it shall be a test case error. If no Default Expression is specified for an Encoding Variation, it is equivalent to the value FALSE being specified. If no Default Expressions are specified or if all evaluate to FALSE, the first Encoding Variation shall be taken as the default.

Encoding variations shall be provided in the format shown in Proforma 32, below.

Encoding Variations			
Group : *[EncVariationGroupReference]*			
Encoding Rule Name : *EncodingRuleIdentifier*			
Type List : *[TypeList]*			
Comments : *[FreeText]*			
Encoding Variation	**Reference**	**Default**	**Comments**
⋮ *EncVariationId&ParList* ⋮	⋮ *VariationReference* ⋮	⋮ *[DefaultExpression]* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*			

Proforma 32 – Encoding Variations

SYNTAX DEFINITION:

463 EncVariationGroupreference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {EncVariationGroupIdentifier "/"}
452 EncodingRuleIdentifier ::= Identifier
467 TypeList ::= Type {Comma Type}
470 EncVariationId&ParList ::= EncVariationIdentifier [FormalParList]
473 Variationreference ::= BoundedFreeText
456 DefaultExpression ::= Expression

EXAMPLE 34 – Encoding Variations:

Encoding Variations			
Encoding Rule Name : BER			
Type List : Length			
Comments : Length is defined to be an INTEGER type.			
Encoding Variation	**Reference**	**Default**	**Comments**
SD LD(len: INTEGER)	6.3.3.1 6.3.3.2	TRUE	
Detailed Comments:			

11.16.3 Invalid Field Encoding Definitions

In order to test encoding rules thoroughly, it may be necessary to define illegal variations of the encoding definitions used by the protocol. Invalid field encoding definitions may be provided for any of the Types used in PDU fields in the test suite. Once defined, an invalid field encoding definition may be used to override the normal encoding of a specific PDU Constraint field value of the same Type (see 13.4).

The following information relative to an invalid field encoding definition shall be provided:

a) an Invalid Field Encoding Name, which is a unique identifier to be used throughout the test suite to refer to this invalid field encoding definition, followed by an optional formal parameter list;

b) an optional Type List, to list the types to which this encoding may be applied; an empty list means that the encoding definition may be applied to any field of a PDU;

c) an Encoding Operation Definition which contains the definition of how the values are to be encoded,

   which shall consist of a procedural definition, in the same form as a procedural definition of a Test Suite Operation (see 11.3.4), which when evaluated results in the evaluation of a ReturnValue statement to provide the result of the operation, including explanatory comments embedded within the procedural definition at appropriate places as text delimited by "/*" and "*/"; explanatory comments shall include an example showing an invocation; the result of the Encoding Operation shall be a Bitstring with a defined order of transmission, being the encoding of the relevant value;

d) optionally, further comment describing the operation, provided either in the Comments part of the table header or in the Detailed Comments area of the table.

The use of procedural definitions is recommended in order to provide precision in the definition of the operations.

If a formal parameter list is specified, the values passed to the encoding operation are used to affect the encoding of the PDU field. Each formal parameter shall be declared to be a Predefined Type, a Test Suite Type Identifier or a PDU Type Identifier. For example, an integer value may be passed to an encoding operation that calculates the length of a PDU field. The way in which parameters passed to the operation are used shall be explained in the encoding operation definition.

One proforma shall be used for each Invalid Field Encoding Definition.

Invalid Field Encoding Operation Definitions shall be provided in Proforma 33, below.

Invalid Field Encoding Operation Definition	
Group	**:** *[InvalidFieldEncodingGroupReference]*
Operation Name	**:** *InvalidFieldEncodingId&ParList*
Result Type	**:** *[TypeList]*
Comments	**:** *[FreeText]*
Definition	
TS_OpProcDef	
Detailed Comments:	*[FreeText]*

Proforma 33 – Invalid Field Encoding Operation Definition

SYNTAX DEFINITION:

484 InvalidFieldEncodingGroupReference:: =
[(SuiteIdentifier | TTCN_ModuleIdentifier)"/"](InvalidFieldEncodingGroupIdentifier"/")

481 InvalidFieldEncodingId&ParList ::= InvalidFieldEncodingIdentifier [FormalParList]

467 TypeList ::= Type {Comma Type}

162 TS_OpProcDef ::= [VarBlock]ProcStatement

11.16.4 Application of encoding rules

Encoding rules specified in the test suite are applied to all PDUs sent or received in the Behaviour Part. Encoding rules may be specified for the whole test suite or for type declarations or constraint declarations, as noted in Table 4. The places in Table 4 marked ✪ identify the allowed scope of application of each of the kinds of encoding information.

Table 4/X.292 – Applicability of Encoding Definitions

		Encoding Definitions				
		Encoding Rules		**Encoding Variations**		**Invalid Field Encodings**
Precedence	**Scope of Application**	**Default**	**Other**	**Default**	**Other**	
Lowest	Test Suite	✪		✪		
	Type Declarations					
	PDUs		✪	✪	✪	
	Structured Types or ASN.1 Types			✪	✪	
	Simple types or PDU fields/elements			✪	✪	✪
	Constraint Declarations					
	PDUs		✪	✪	✪	
	Structured Types or ASN.1 Types			✪	✪	
Highest	PDU fields/ elements			✪	✪	✪
Precedence within a row		Lowest				Highest

The encoding rules shall be applied according to the precedence values of the rows shown in the first column in Table 4, with "(4)" having the highest priority, and "(1)" having the lowest. Within each row, the precedence is from left to right, with the rightmost entry having the highest precedence. Thus, Constraint field encoding rules have precedence over all others, while default encoding rules applied at the test suite level may be overridden by any of the other specification methods. The actual encoding rules to be used for a PDU after all overrides have been applied are referred to as the applicable encoding rules.

If no encoding information is specified on a structured or ASN.1 Type Constraint, it inherits the encoding rules applied at the PDU level. Thus, the encoding rules applied to a structured or ASN.1 Type Constraint will vary, based on the PDU in which it is used. Conversely, if encoding information is specified on a Structured or ASN.1 Type Constraint, it will override the encoding information of every PDU in which it is used. If such a Structured or ASN.1 Type Constraint is used in an ASP, the encoding information is ignored.

On RECEIVE events, if no specific encoding rules apply to the incoming PDU, it can be encoded in any variation allowed by the applicable Encoding Definition (e.g. any form of length encoding allowed by BER).

11.17 CM Type Definitions

11.17.1 Introduction

CM parameters may be of any type that may be specified in TTCN. Simple CMs may contain no associated parameters or may contain just one parameter, e.g. a natural number, a preliminary result, or a character string like "suspend" or "continue". More complex CMs may carry additional information, e.g. a whole PDU, a PDU field, or the value read from a timer. There are no predefined CMs.

11.17.2 CM Type Definitions using tables

CM Types may be declared using TTCN tables. The following information shall be provided for each CM type:

a) its name,

   where each name shall be unique within the test suite;

b) a list of parameters associated with the CM,

   where the following information shall be provided for each parameter:

   1) its name,

      which shall be unique within the CM;

   2) its type and an optional attribute,

      in the same way as for PDU fields,

      in which case the specification may restrict the field to a particular length or a range according to 11.18. The length values shall be interpreted according to Table 5 in 11.18. The boundaries shall be specified in terms of non-negative INTEGER literals, Test Suite Parameters, Test Suite Constants or the keyword INFINITY.

      The length specifications defined for the PDU field type in the Test Suite Type definitions shall not conflict with the length specifications in the PDU type definition, i.e., the set of strings defined by a length restriction in a PDU definition shall be a true subset of the set of strings defined by the Test Suite Type definition.

      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.

      All parameters of CMs are optional, that is they may be omitted when the CM is used.

This information shall be provided in the format shown in Proforma 34, below.

CM Type Definition		
CM Name : *CM_Identifier*		
Group : *[CM_GroupReference]*		
Comments : *[FreeText]*		
Parameter Name	**Parameter Type**	**Comments**
⋮ *CM_ParIdOrMacro* ⋮	⋮ *Type&Attributes* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*		

Proforma 34 – CM Type Definition

The Parameter Name and Parameter Type columns shall either be both present or both omitted.

SYNTAX DEFINITION:

424 CM_Identifier ::= Identifier

426 CM_GroupReference ::= [(SuiteIdentifier|TTCN_ModuleIdentifier) "/"] {CM_GroupIdentifier "/"}

431 CM_ParIdOrMacro ::= CM_ParIdentifier|MacroSymbol

396 Type&Attributes ::= (Type[LengthAttribute]) |PDU

11.17.3 CM Type Definitions using ASN.1

CM Types may be declared using ASN.1. The following information shall be provided for each ASN.1 CM type:

a) its name,

   where each name shall be unique within the test suite;

b) the ASN.1 CM type definition,

   which shall follow the syntax defined in Recommendations X.680. For identifiers within that definition the hyphen symbol (-) shall not be used. The underscore symbol (_) may be used instead. The PDU identifier in the table header is the name of the first type defined in the table body.

Types referred to from the PDU definition shall be defined in other ASN.1 type definition tables, be defined by reference in the ASN.1 type reference table or be defined locally in the same table, following the first type definition. Locally defined types shall not be used in other parts of the test suite.

ASN.1 comments can be used within the table body. The comments column shall not be present in this table.

Comments in ASN.1 start with "--" and end with either the next occurrence of "--" or with "end of line", whichever comes first. This prevents a single ASN.1 comment from spanning several lines. "End of line" is not, however, a defined symbol in TTCN.MP. ATS specifiers are recommended to facilitate the exchange of ATSs in TTCN.MP by always closing ASN.1 comments with "--".

This information shall be provided in the format shown in Proforma 35, below.

ASN.1 CM Type Definition	
CM Name	**:** *CM_Identifier*
Group	**:** *[ASN1CM_GroupReference]*
Comments	**:** *[FreeText]*
Type Definition	
ASN1_Type&LocalTypes	
Detailed Comments:	*[FreeText]*

Proforma 35 – ASN.1 CM Type Definition

SYNTAX DEFINITION:

424 CM_Identifier ::= Identifier

440 ASN1_CM_GroupReference ::= [(SuiteIdentifier|TTCN_ModuleIdentifier) "/"]{ASN1_CM_GroupIdentifier "/"}

121 ASN1_Type&LocalTypes ::= ASN1_Type{ASN1_LocalType}

11.18 String length specifications

11.18.1 TTCN permits the specification of length restrictions on string types (i.e., BITSTRING, HEXSTRING, OCTETSTRING and all CharacterString types, plus the ASN.1 types BIT STRING and OCTET STRING) in the following instances:

a) when declaring Test Suite Types as a type restriction;

b) when declaring simple ASP parameters, PDU fields and elements of Structured Types as an attribute of the parameter, field or element type;

c) when defining ASP/PDU or Structured Type constraints as an attribute of the constraint value.

11.18.2 Length specifications can have the following formats:

a) [Length]

restricting the length of the possible string values of a type to exactly *Length*;

b) [MinLength **TO** MaxLength] or [MinLength .. MaxLength]

specifying a minimum and a maximum length for the values of a particular string type.

The length boundaries: *Length*, *MinLength* and *MaxLength* are of different complexity depending on where they are used. In all cases, these boundaries shall evaluate to non-negative INTEGER values. For the upper bound the keyword INFINITY may also be used to indicate that there is no upper limit for the length. Where a range length is specified, the lower of the two values shall be specified on the left.

In the context of constraints, length restrictions can also be specified on values of type SEQUENCE OF or SET OF, thus limiting the number of their elements.

Table 5 specifies the units of length for different string types.

Table 5/X.292 – Units of length used in field length specifications

Type	Units of Length
BITSTRING or BIT STRING	Bits
HEXSTRING	Hex digits
OCTETSTRING or OCTET STRING	Octets
CharacterString	Characters
SEQUENCE OF	Elements of its base type
SET OF	Elements of its base type

Length specifications shall not conflict, i.e., a restriction on a type (set of values) that is already restricted shall specify a subrange of values of its base type.

EXAMPLE 35 – Length specification

Assume the following ASN.1 type definitions:

type1 ::= OCTETSTRING [0 .. 25]

type2 ::= type1 [15 .. 24]

The length restriction on type2 is correct since type2 comprises all OCTETSTRING values having a minimum length of 15 and a maximum length of 24, which is a true subset of all OCTETSTRINGs of a maximum length of 25. On the other hand:

type2 ::= type1[15 .. 30]

is invalid since it contains values not included in type1.

11.19 ASP, PDU and CM Definitions for SEND events

In ASPs and/or PDUs that are sent from the tester, values for ASP parameters and/or PDU fields that are defined in the Constraints Part (see clauses 12, 13 and 14) shall correspond to the parameter or field definition. This means:

a) the value shall be of the type specified for that ASP parameter or PDU field; and

b) each value shall satisfy any relevant length restrictions associated with the type;

c) PDU field values shall be encoded in accordance with applicable encoding rules.

The encoding operations defined in the test suite are performed implicitly as part of the SEND event. Defaults and overrides are applied, as necessary. Thus, the output of the SEND event is the encoded data to be passed to the relevant service provider.

11.20 ASP, PDU and CM Definitions for RECEIVE events

For ASPs and/or PDUs received by the tester the ASPs and/or PDU type defines the class of incoming ASPs and/or PDUs that can match an event specification of that type. An incoming ASP or PDU is considered to be of that class if and only if:

a) the ASP parameter and/or PDU field values are of the type specified in the ASP and/or PDU definition; and

b) the value satisfies any relevant length restrictions associated with the type;

c) PDU field values can be decoded in accordance with applicable encoding rules.

In all other cases an incoming ASP and/or PDU does not match an event specification of that type.

In the case of substructured ASPs and/or PDUs, either using Structured Types or ASN.1, the above rules apply to the fields of the substructure(s) recursively.

11.21 Alias Definitions

11.21.1 Introduction

In order to enhance the readability of TTCN behaviour descriptions, an Alias may be used to facilitate the renaming of ASP and/or PDU identifiers in behaviour descriptions. This renaming may be done to highlight the exchange of PDUs embedded in ASPs.

The following information shall be provided for each Alias:

a) an Alias identifier;

b) its expansion,

which is itself an identifier.

This information shall be provided in format shown in Proforma 36, below.

Alias Definitions		
Group : *[AliasGroupReference]*		
Alias Name	**Expansion**	**Comments**
⋮ *AliasIdentifier* ⋮	⋮ *Expansion* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*		

Proforma 36 – Alias Definitions

Collective comments may be used in this table according to Figure 2.

SYNTAX DEFINITION:

493 AliasGroupReference ::= [(SuiteIdentifier │TTCN_ModuleIdentifier) "/"] {AliasGroupIdentifier "/"}

497 Alias Identifier ::= Identifier

499 Expansion ::=ASP_Identifier│PDU_Identifier

11.21.2 Expansion of Aliases

The following rules shall apply:

a) An Alias is an identifier that shall follow the syntax rules for identifier defined in the TTCN.MP. This means that an Alias is delimited by any character (symbol) not allowed in a TTCN identifier.

b) Aliases are not transitive – If one Alias appears as the expansion of another Alias it shall not be expanded (i.e., it is a one pass expansion).

c) An Alias shall be used only to replace an ASP identifier or a PDU identifier within a single TTCN statement in a behaviour tree. It shall be used only in a behaviour description column.

d) The expansion of an Alias shall follow the syntax rules for identifier as defined in the TTCN.MP.

EXAMPLE 36 – Alias definition from a Transport Test Suite:

Alias Definitions		
Alias Name	**Expansion**	**Comments**
CR	N_DATArequest	Alias for the N_DATArequest ASP Used to carry a CR_TPDU
DR	N_DATArequest	Alias for the N_DATArequest ASP Used to carry a DR_TPDU
CC	N_DATAindication	Alias for the N_DATAindication ASP Used to carry a CC_TPDU

NOTE – Because Aliases are treated as macro expansions, the term AliasIdentifier does not appear in the BNF for TTCN event lines.

12 Constraints Part

12.1 Introduction

An ATS shall specify the values of the ASP parameters and PDU fields that are to be sent or received by the test system. The constraints part fulfils that purpose in TTCN.

The dynamic behaviour descriptions (see clause 15) shall reference constraints to construct outgoing ASPs and/or PDUs in SEND events; and to specify the expected contents of incoming ASPs and/or PDUs in RECEIVE events.

Constraints can be specified in either of the two forms:

a) tabular constraints (see clause 13);

b) ASN.1 constraints (see clause 14).

Actual values or constraints on the values of a CM shall be declared in the same way as PDU constraints are to be declared.

12.2 General principles

This subclause describes the general principles and defines the mechanisms of how to build constraints for SEND events and how to match RECEIVE events. These principles are common to both the tabular and ASN.1 forms of constraints.

Constraints are detailed specifications of ASPs and/or PDUs. Normally, each constraint is defined specifically for use with either SEND events or RECEIVE events. A constraint need not be specified if an ASP or CM has not parameters or if PDU has no fields. Any given constraint may be used in either context, provided the operational semantic restrictions defined in Annex B are met.

The constraint specification of an ASP and/or PDU shall have the same structure as that of the type definition of that ASP or PDU.

If an ASP and/or PDU is substructured, then the constraints for ASPs and/or PDUs of that type shall have the same tabular structure or a compatible ASN.1 structure (i.e., possibly with some groupings).

Structured Types expanded into an ASP or PDU definition by use of the macro symbol (<-) are not considered to be substructures. Constraints for such ASPs or PDUs shall either have a completely flat structure (i.e., the elements of an expanded structure are explicitly listed in the ASP or PDU constraint) or shall reference a corresponding structure constraint for macro expansion.

Constraints specify ASP parameter and PDU field values using various combinations of literal values, data object references, expressions, ASN.1 constructed values, special matching mechanisms and references to other constraints. Constraints applying to the whole of or part of a PDU may also specify encoding rules to override the general encoding rules being applied in the test suite. Such encoding rules may be specified for the whole Constraint or for a single field of the Constraint.

Values of all TTCN or ASN.1 types can be used in constraints. Expressions used in constraints shall evaluate to a specific value when the constraint is used for sending or receiving events.

Whichever way the values are obtained, they shall correspond to the parameter or field entries in the ASP or PDU type definitions. This means:

a) the value shall be of the type specified for that parameter or field; and

b) the length shall satisfy any restriction associated with the type.

An expression in a constraint shall contain only Values (including, for example, ConstraintValue&Attributes), Test Suite Parameters, Test Suite Constants, formal parameters, Component References and Test Suite Operations.

A constraint reference (possibly parameterized) is also allowed as a parameter or field value (static chaining).

Neither Test Suite Variables nor Test Case Variables shall be used in constraints, unless passed as actual parameters. In the latter case they shall be bound to a value and are not changed by the occurrence of a SEND or a RECEIVE event.

Matching mechanisms are defined in 12.6.2.

12.3 Parameterization of constraints

Constraints may be parameterized. In such cases the constraint name shall be followed by a formal parameter list enclosed in parentheses. The formal parameters shall be used to specify ASP parameter or PDU field values in the constraint.

Each formal parameter name shall be followed by a colon and the name of the parameter type. If more than one parameter of the same type is used, the parameter may be specified as a parameter sublist. When a parameter sublist is used, the parameter names shall be separated by a comma. The final parameter in the sublist shall be followed by a colon and the name of the parameter sublist type. When more than one parameter and type pair (or parameter sublist and type pair) is used, the pairs shall be separated from each other by semicolons.

Literal values, Test Suite Parameters, Test Suite Constants, Test Suite Variables, Test Case Variables and PDU or Test Suite Type constraints may be passed as actual parameters to a constraint in a constraints reference made from a behaviour description. The parameters shall not be of PCO type or ASP type.

12.4 Chaining of constraints

Constraints may be chained by referencing a constraint as the value of a parameter or field in another constraint. For example, the value of the Data parameter of an N-DATAreq (Network Data Request) ASP could be a reference to a T-CRPDU (Transport Connect Request PDU) PDU constraint, i.e., the T-CRPDU is chained to the N-DATAreq ASP.

Constraints can be chained in one of two ways, either by:

a) static chaining, where an ASP parameter value or PDU field value in a constraint is an explicit reference to another constraint; or

b) dynamic chaining, where an ASP parameter value or PDU field value in a constraint is a formal parameter of the constraint. When such a constraint is referenced from a dynamic behaviour, the corresponding actual parameter to the constraint is a reference to another constraint (see Appendix I for examples of static and dynamic chaining).

Wherever constraints are referenced within constraints declarations, those references shall not be recursive (neither directly or indirectly).

Chaining of constraints may only be used if the appropriate declarations have been set up to allow chaining. For example, if an ASP parameter is to be chained to a PDU constraint, then the ASP parameter shall be declared to be of an appropriate PDU type or the meta-type **PDU.** In ASN.1 PDU declarations, the PDU type might well be one defined as a CHOICE of all valid individual PDU types, whereas in tabular PDU declarations the meta-type **PDU** would need to be used to achieve a similar effect. Similarly, if a PDU field is to be chained to a Structure constraint, then the PDU field shall be declared to be of an appropriate Structure type.

12.5 Constraints for SEND events

Constraints that are referenced for SEND events shall not include wildcards [i.e., AnyValue (?) or AnyOrOmit (*)] unless these are explicitly assigned specific values on the SEND event line in the behaviour description.

In tabular constraints, all ASP parameters and PDU fields are optional and therefore may be omitted using the Omit symbol, to indicate that the ASP parameter or PDU field is to be absent from the event sent.

In ASN.1 constraints, only ASP parameters and PDU fields declared as OPTIONAL may be omitted. These may be omitted either by using the Omit symbol or by simply leaving out the relevant ASP parameter or PDU field.

None of the matching mechanisms defined in 12.6.2, except SpecificValue, provides a value for an ASP parameter or PDU field on a SEND event.

In cases where ASN.1 values of type SET or SET OF are used in a constraint, the values of the elements of the set shall be sent in the order specified by the relevant constraint.

12.6 Constraints for RECEIVE events

12.6.1 Matching values

If a constraint is to be used to construct the values of ASP parameters or PDU fields that a received ASP or PDU shall match, it shall contain only specific values evaluated as explained in 12.6.3, or special matching mechanisms where it is not desirable, or possible, to specify specific values. The matching mechanisms specify other ways of matching than "equal to a specific value".

An incoming ASP and/or PDU matches a constraint used in a RECEIVE event if and only if all the following conditions are met:

a) all the ASP parameters and/or PDU fields are of the type specified in the ASP and/or PDU definitions;

b) the value, alphabet and length satisfies any restriction associated with the type;

c) the ASP parameter and/or PDU field values correctly match those of the constraint;

d) for PDUs, the correct decoding of the PDU has taken place, taking into account applicable encoding rule defaults and overrides; if encoding rules other than those specified for the constraint have been used to encode the received PDU, then that received PDU will not match.

In the case of substructured ASPs and/or PDUs, either using Structured Types or ASN.1, the above rules shall apply to the fields of the substructure(s) recursively.

NOTE – If a RECEIVE event is qualified by a Boolean expression, then a successful match means that both the incoming ASP and/or PDU must match the constraint and that the qualifier must evaluate to TRUE.

12.6.2 Matching mechanisms

An overview of the supported matching mechanisms is shown in Table 6, including the special symbols and the scope of their application. The left hand column of this table lists all the ASN.1 types and TTCN equivalent types to which these matching mechanisms apply. The matching mechanisms in the horizontal headings are arranged in four groups:

a) specific values;

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

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

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

Some of the symbols may be used in combination, as detailed in the following subclauses.

The shaded area in Table 6 indicates the mechanisms that apply to both predefined TTCN and ASN.1 types.

In a constraint specification, the matching mechanisms may replace values of single ASP parameters or PDU fields or even the entire contents of an ASP or PDU.

NOTE – When these matching mechanisms are used singly or in combination, many protocol restrictions can be specified in the constraints, thereby avoiding undesirable computation details in the behaviour part.

12.6.3 Specific Value

This is the basic matching mechanism. Specific values in constraints are expressions. Unless otherwise specified, a constraint ASP parameter or PDU field matches the corresponding incoming ASP parameter or PDU field if and only if the incoming ASP parameter or PDU field has exactly the same value as the value to which the expression in the constraint evaluates.

Two values of a tabular ASP, PDU or Structured Type, or of ASN.1 SEQUENCE or SEQUENCE OF are considered the same if each of their parameters fields or elements match and are in the same order. For ASN.1 SET and SET OF types two values are the same if they have the same number of elements, and each element in one value matches exactly one element in the other value. The elements in a SET or SET OF type value need not be in the same order to match.

Table 6/X.292 – TTCN Matching Mechanisms

	VALUE	INSTEAD OF VALUE								INSIDE VALUE			ATTRIBUTES	
TYPE	**Specific Value**	**Complement**	**Omit(-)**	**AnyValue (?)**	**AnyOrOmit(*)**	**ValueList**	**Range**	**SuperSet**	**SubSet**	**AnyOne (?)**	**AnyOrNone(*)**	**Permutation**	**Length**	**IfPresent**
BOOLEAN	•	•	•	•	•	•								•
INTEGER	•	•	•	•	•	•	•							•
ENUMERATED	•	•	•	•	•	•								•
BITSTRING	•	•	•	•	•	•				•	•		•	•
OCTETSTRING	•	•	•	•	•	•				•	•		•	•
HEXSTRING	•	•	•	•	•	•				•	•		•	•
CHARSTRNGS	•	•	•	•	•	•				•	•		•	•
SEQUENCE		•	•	•	•	•								•
SEQUENCE OF	•	•	•	•	•	•				•	•	•	•	•
SET	•	•	•	•	•	•								•
SET OF	•	•	•	•	•	•		•		•	•		•	•
ANY	•	•	•	•	•	•								•
CHOICE	•	•	•	•	•	•								•
OBJECT ID	•	•	•	•	•	•								•

12.6.4 Instead of Value

12.6.4.1 Complement

Complement is an operation for matching that can be used on all values of all types. Complement is denoted by the keyword COMPLEMENT followed by a list of constraint values. Each constraint value in the list shall be of the type declared for the ASP parameter or PDU field in which the Complement mechanism is used.

SYNTAX DEFINITION:

566 Complement ::= **COMPLEMENT** ValueList

A constraint ASP parameter or PDU field that uses Complement matches the corresponding ASP parameter or PDU field if and only if the incoming ASP parameter or PDU field does not match any of the values listed in the ValueList.

EXAMPLE 37 – Constraints using Complement instead of a value, and with a value list:

Type	*Constraint*
INTEGER	COMPLEMENT(5)
INTEGER	COMPLEMENT(1, 3, 5)

12.6.4.2 Omit

Omit is a special symbol for matching that can be used on values of all types, provided that the ASP parameter or PDU field is optional.

In ASN.1 constraints it is also possible to simply leave out an OPTIONAL ASP parameter or PDU field instead of using OMIT explicitly.

NOTE – In tabular constraints, all parameters, fields and elements are considered to be implicitly optional, and hence may be omitted using Omit. In ASN.1 constraints, parameters, fields and elements which are not explicitly marked as OPTIONAL in the type definition are mandatory and cannot be omitted without violating the type definition. If such a parameter, field or element needs to be omitted from a particular constraint, either another type needs to be defined in which that parameter, field or element is explicitly marked as OPTIONAL (perhaps by marking everything as OPTIONAL), or an Invalid Field Encoding needs to be applied to that parameter, field or element, with the effect of omitting it from the encoding.

In tabular constraints Omit shall be denoted by hyphen (-). In ASN.1 constraints Omit is denoted by **OMIT**.

SYNTAX DEFINITION:

567 Omit ::= Dash | **OMIT**

An Omit symbol in a constraint is used to indicate that an optional ASP parameter or PDU field shall be absent.

EXAMPLE 38 – Constraint using Omit instead of a value, at top level:

Type	*Constraint*
INTEGER OPTIONAL	OMIT

12.6.4.3 AnyValue

AnyValue is a special symbol for matching that can be used on values of all types. In both tabular and ASN.1 constraints AnyValue is denoted by "?".

SYNTAX DEFINITION:

568 AnyValue ::= "?"

A constraint ASP parameter or PDU field that uses AnyValue matches the corresponding incoming ASP parameter or PDU field if and only if the incoming ASP parameter or PDU field evaluates to a single element of the specified type.

EXAMPLE 39 – Constraint using Value in combination with AnyValue:

Type	*Constraint*
SEQUENCE OF SET OF INTEGER	{ {1, 2}, ?, {1, 2, ?} }

12.6.4.4 AnyOrOmit

AnyOrOmit is a special symbol for matching that can be used on values of all types, provided that the ASP parameter or PDU field is declared as optional. In both tabular and ASN.1 constraints AnyOrOmit is denoted by "*".

NOTE – The symbol "*" is used for both AnyOrOmit and AnyOrNone. Ambiguity in interpretation is resolved by the requirements in this subclause and 12.6.5.2.

SYNTAX DEFINITION:

569 AnyOrOmit ::= "*"

A constraint ASP parameter or PDU field that uses AnyOrOmit matches the corresponding incoming ASP parameter or PDU field if and only if either the incoming ASP parameter or PDU field evaluates to any element of the specified type, or if the incoming ASP parameter or PDU field is absent.

EXAMPLE 40 – Constraint using Value in combination with AnyOrOmit:

Type	*Constraint*
SEQUENCE OF { id1 SET OF INTEGER, id2 SET OF INTEGER }	{ id1 {2, 5}, id2 * }

12.6.4.5 ValueList

ValueList can be used on values of all types. In both tabular and ASN.1 constraints. ValueLists are denoted by a parenthesized list of values separated by commas.

SYNTAX DEFINITION:

570 ValueList ::= "("ConstraintValue&Attributes {Comma ConstraintValue&Attributes} ")"

A constraint ASP parameter or PDU field that uses a ValueList matches the corresponding incoming ASP parameter or PDU field if and only if the incoming ASP parameter or PDU field value matches any one of the values in the ValueList. Each value in the ValueList shall be of the type declared for the ASP parameter or PDU field in which the ValueList mechanism is used.

EXAMPLE 41 – Constraint using ValueList instead of a specific value, for INTEGER type:

Type	*Constraint*
INTEGER	(2, 4, 6)

EXAMPLE 42 - Constraints using ValueList instead of a specific value, for CHOICE type:

Type	*Constraint*
CHOICE { a INTEGER, b BOOLEAN }	(a 2, b TRUE)

12.6.4.6 Range

Ranges shall be used only on values of INTEGER type. A range is denoted by two boundary values, separated by ".." or TO, enclosed by parentheses. A boundary value shall be either:

a) INFINITY or -INFINITY;

b) a constraint expression that evaluates to a specific INTEGER value.

The lower boundary shall be put on the left side of the ".." or TO, the upper boundary at the right side. The lower boundary shall be less than the upper boundary.

SYNTAX DEFINITION:

571 ValueRange ::= "(" ValRange ")"
572 ValRange ::= (LowerRangeBound To UpperRangeBound)
573 LowerRangeBound ::= ConstraintExpression | Minus **INFINITY**
574 UpperRangeBound ::= ConstraintExpression | **INFINITY**

A constraint ASP parameter or PDU field that uses a Range matches the corresponding incoming ASP parameter or PDU field if and only if the incoming ASP parameter or PDU field value is equal to one of the values in the Range.

EXAMPLE 43 – Constraint using Range instead of a value:

Type	*Constraint*
INTEGER	(1 .. 6) (-INFINITY .. 8) (12 .. INFINITY)

12.6.4.7 SuperSet

SuperSet is an operation for matching that shall be used only on values of SET OF type. SuperSet shall be used only in ASN.1 constraints. SuperSet is denoted by **SUPERSET**.

SYNTAX DEFINITION:

575 SuperSet ::= SUPERSET "(" ConstraintValue&Attributes ")"

A constraint ASP parameter or PDU field that uses SuperSet matches the corresponding incoming ASP parameter or PDU field if and only if the incoming ASP parameter or PDU 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 ASP parameter or PDU field in which the SuperSet mechanism is used.

EXAMPLE 44 – Constraint using SuperSet instead of a specific value:

Type	*Constraint*
SET OF INTEGER	SUPERSET({1, 2, 3})

12.6.4.8 SubSet

SubSet is an operation for matching that can be used only on values of SET OF type. SubSet shall be used only in ASN.1 constraints. SubSet is denoted by **SUBSET**.

SYNTAX DEFINITION:

576 SubSet ::= SUBSET "(" ConstraintValue&Attributes")"

A constraint ASP parameter or PDU field that uses SubSet matches the corresponding incoming ASP parameter or PDU field if and only if the incoming ASP parameter or PDU field contains only elements defined within the SubSet, and may contain less. The argument of SubSet shall be of the type declared for the ASP parameter or PDU field in which the SubSet mechanism is used.

EXAMPLE 45 – Constraint using SubSet instead of a specific value:

Type	*Constraint*
SET OF INTEGER	SUBSET({2, 4, 6, 8, 10})

12.6.5 Inside Values

12.6.5.1 AnyOne

AnyOne is a special symbol for matching that can be used within values of string types, SEQUENCE OF and SET OF. In both tabular and ASN.1 constraints AnyOne is denoted by "?".

SYNTAX DEFINITION:

754 AnyOne ::= "?"

Inside a string, SEQUENCE OF or SET OF a "?" in place of a single element means that any single element will be accepted. If the symbol "?" is needed within a CharacterString as a character, it shall be indicated by "\?". If the symbol "\" is needed within a CharacterString as a character, it shall be indicated by "\\".

EXAMPLE 46 – Constraints using AnyOne:

Type	*Constraint*
IA5String	"a?cd"
SEQUENCE OF INTEGER	{1, 2, ? }

NOTE – The "?" in the second example can be interpreted as an AnyValue replacing an INTEGER value, or AnyOne inside a SEQUENCE OF INTEGER value. Since both interpretations lead to the same set of events that match the constraint, no problem arises.

12.6.5.2 AnyOrNone

AnyOrNone is a special symbol for matching that can be used within values of string types, SEQUENCE OF and SET OF. In both tabular and ASN.1 constraints AnyOrNone is denoted by "*".

If a "*" appears at the highest level inside a value of string type, SEQUENCE OF or SET OF, it shall be interpreted as AnyOrNone.

NOTE – This rule prevents the otherwise possible interpretation of "*" as AnyOrOmit that replaces an element inside the string, SEQUENCE OF or SET OF.

SYNTAX DEFINITION:

755 AnyOrNone ::= "*"

Inside a string, SEQUENCE OF or SET OF a "*" in place of a single element means that either none, or any number of consecutive elements will be accepted. The "*" symbol matches the longest sequence of elements possible, according to the pattern as specified by the symbols surrounding the "*". If the symbol "*" is needed within a CharacterString as a character, it shall be indicated by "*". If the symbol "\" is needed within a CharacterString as a character, it shall be indicated by "\\".

EXAMPLE 47 – Constraints using AnyOne:

Type	*Constraint*
IA5String	"ab*z"
SEQUENCE OF INTEGER	{1, 2, *, 10 }
SEQUENCE OF IA5String	{ "ab*z", *, "abc" }

12.6.5.3 Permutation

Permutation is an operation for matching that can be used only on values inside a value of SEQUENCE OF type. Permutation shall be used only in ASN.1 constraints. Permutation is denoted by **PERMUTATION**.

SYNTAX DEFINITION:

577 Permutation ::= **PERMUTATION** ValueList

Permutation in place of a single element means that any series of elements is acceptable provided it contains the same elements as the value list in the Permutation, though possibly in a different order. If both Permutation and AnyOrNone are used inside a value, the AnyOrNone shall be evaluated first. Each element listed in Permutation shall be of the type declared inside the SEQUENCE OF type of the ASP parameter or PDU field.

EXAMPLE 48 – Constraint using Permutation:

Type	*Constraint*
SEQUENCE OF INTEGER	{PERMUTATION (1, 2, 3), 5}

EXAMPLE 49 - Constraints using Permutation in combination with AnyOrNone:

Type	*Constraint*
SEQUENCE OF INTEGER	{PERMUTATION (1,2,3), *}
	{PERMUTATION (1,2,3,*)}

Note that the first constraint matches with incoming ASPs and/or PDUs that consist of a sequence of INTEGER values, starting with 1,2,3; 1,3,2; 2,1,3; 2,3,1; 3,1,2; or 3,2,1 and followed by any number of values of type INTEGER. The second constraint matches any incoming ASP and/or PDU of type SEQUENCE OF INTEGER, that contains the elements 1, 2,3 in any order and in any position. It matches, for example; {5,2,7,1,3} and {9,3,7,2,12,1,17}.

12.6.6 Attributes of values

12.6.6.1 Length

Length is an operation for matching that can be used only as an attribute of the following mechanisms: Complement, AnyValue, AnyOrOmit, AnyOne, AnyOrNone, Permutation, SuperSet and SubSet. It can be used in conjunction with the IfPresent attribute.

In both tabular and ASN.1 constraints, length may be specified as an exact value or range in string values and SEQUENCE OF or SET OF values, according to 11.18. The units of length are to be interpreted according to Table 5. The boundaries shall be denoted by 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 defined for the ASP parameter or PDU field type in the Test Suite Type definitions shall not conflict with the length specifications in the ASP or PDU constraint, i.e., the set of strings defined by a length restriction in an ASP or PDU constraint shall be a true subset of the set of strings defined by the ASP or PDU definition.

SYNTAX DEFINITION:

580 ValueLength ::= SingleValueLength | RangeValueLength

581 SingleValueLength ::= "[" ValueBound "]"

582 ValueBound ::= Number | TS_ParIdentifier | TS_ConstIdentifier | FormalParIdentifier

583 RangeValueLength ::= "[" LowerValueBound To UpperValueBound "]"

584 LowerValueBound ::= ValueBound

89 To ::=TO |".."

585 UpperValueBound ::= ValueBound | **INFINITY**

A constraint ASP parameter or PDU field that uses Length as an attribute of a symbol matches the corresponding incoming ASP parameter or PDU field if and only if the incoming ASP parameter or PDU field matches both the symbol and its associated attribute. The length attribute matches if the length of the incoming ASP parameter or PDU 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 ASP parameter or PDU field is exactly the specified value.

In the case of an omitted parameter, field or element, Length is always considered as matching. Hence, with Omit it is redundant and with AnyOrOmit and IfPresent it places a restriction on the incoming value, if any.

EXAMPLE 50 – Constraints using Value in combination with Length:

Type	*Constraint*
IA5String	"ab*ab" [13]

12.6.6.2 IfPresent

IfPresent is a special symbol for matching that can be used as an attribute of all the matching mechanisms, provided the type is declared as optional. In both tabular and ASN.1 constraints IfPresent is denoted by **IF_PRESENT**.

A constraint ASP parameter or PDU field that uses an IfPresent symbol as an attribute of another symbol matches the corresponding incoming ASP parameter or PDU field if and only if the incoming ASP parameter or PDU field matches the symbol, or if the incoming ASP parameter or PDU field is absent.

NOTE – The AnyOrOmit symbol (*) has exactly the same meaning as ? IF_PRESENT

EXAMPLE 51 – Constraints using Value in combination with IfPresent:

Type	*Constraint*
IA5String OPTIONAL	"abcdef" IF_PRESENT

13 Specification of constraints using tables

13.1 Introduction

This clause describes the specification of tabular constraints on Structured Types, ASPs and PDUs. It describes how single constraint tables can be used to specify constraints on flat (unstructured) ASPs or PDUs and how structured constraints can be specified by declaring constraints on Structured Types, defined in the Test Suite Types.

In Annex C additional tables are defined which allow many single constraint declarations in a single table.

13.2 Structured Type Constraint Declarations

If an ASP or PDU is defined using Structured Types, either as macro expansions or substructures, constraints for these ASPs or PDUs shall be similarly substructured. The following information shall be supplied for each Structured Type Constraint:

a) The name of the constraint,

   which may be followed by an optional formal parameter list.

b) The structured type name.

c) The derivation path (see 13.6).

d) The Encoding Variations to be used for the Constraint.

   In order to specify explicit Encoding Variations for entire Structured Type Constraints, which override the encoding rules and Encoding Variations applicable to the PDU Constraint in which this Structured Type Constraint is used, this optional entry shall reference an entry in the relevant Encoding Variations table [e.g. to change from SD to LD(3)]. If this entry is not used, then the encoding rules and Encoding Variations applicable to the PDU Constraint apply to this Structured Type Constraint as well. See 11.16.4.

e) A constraint value for each element,

   where the following information shall be supplied for each element:

   1) Its name.

      Each entry in the element name column shall have been declared in the relevant Structured Type definition. If any of the original elements is defined as having both a short name and full identifier, the constraint shall not repeat the full identifier.

If the Structured Type definition refers to another Structured Type by macro expansion (i.e., with "<-" in place of the element name), then in a corresponding constraint either:

– the individual elements from the Structured Type shall be included directly within the constraints; or

– the macro symbol (<-) shall be placed in the corresponding position in the Element Name column of the constraint and the value shall be a reference to a constraint for the Structured Type referenced from this Structured Type's definition.

Use of Structured Constraints by macro expansion in a constraint shall not be used unless the corresponding Structured Type definition also references the inner Structured Type by macro expansion.

2) Its value and an optional attribute.

3) Optionally, a specific encoding identifier followed by any necessary actual parameter list, in order to a specify explicit encoding for the individual element of a Structured Type Constraint, which override the encoding rules and Encoding Variations applicable to the whole Structured Type Constraint, and which also override any encoding specified for this element in the Structured Type declaration; the encoding identifier, if any, shall identify either one of the Encoding Variations or an Invalid Field Encoding Definition defined in the test suite [e.g. LD(10)]; see 11.16.4.

The element values for structure constraints shall be provided in the format shown in Proforma 37, below.

Structured Type Constraint Declaration			
Constraint Name	**:** *ConsId&ParList*		
Group	**:** *[StructTypeConstraintGroupReference]*		
Structured Type	**:** *StructIdentifier*		
Derivation Path	**:** *[DerivationPath]*		
Encoding Variation	**:** *[EncVariationCall]*		
Comments	**:** *[FreeText]*		
Element Name	**Element Value**	**Element Encoding**	**Comments**
⋮ *ElemIdentifier* ⋮	⋮ *ConstraintValue&Attributes* ⋮	⋮ *[PDU_FieldEncodingCall]* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*			

Proforma 37 – Structured Type Constraint Declaration

This proforma is used in the same way that the PDU Constraint Declaration proforma is used for PDUs (see 13.4).

SYNTAX DEFINITION:

555 ConsId&ParList ::= ConstraintIdentifier [FormalParList]

508 StructTypeConstraintGroupReference
::=[(SuiteIdentifier| TTCN_ModuleIdentifier)"/"]{StructTypeConstraintGroupIdentifier"/"}

99 StructIdentifier ::=Identifier

558 DerivationPath ::= {ConstraintIdentifier Dot }+

511 EncVariationCall ::=EncVariationIdentifier [ActualParList]

107 ElemIdentifier ::=Identifier

562 ConstraintValue&Attributes ::=ConstraintValue ValueAttributes

515 PDU_FieldEncodingCall ::= EncVariationCall| InvalidFieldEncodingCall

If an ASP or PDU definition refers to a Structured Type as a substructure of a parameter or field (i.e., with a parameter name or a field name specified for it), then the corresponding constraint shall have the same parameter or field name in the corresponding position in the parameter name or field name column of the constraint and the value shall be a reference to a constraint for that parameter or field (i.e., for that substructure in accordance with the definition of the

Structured Type). If the ASP or PDU definition refers to a parameter or field specified as being of metatype PDU, then in a corresponding constraint the value for that parameter or field shall be specified as the name of a PDU constraint, or formal parameter.

13.3 ASP Constraint Declarations

The parameter values for ASP constraints shall be provided in the format shown in Proforma 38, below.

ASP Constraint Declaration		
Constraint Name : *ConsId&ParList*		
Group : *[ASP_ConstraintGroupReference]*		
ASP Type : *ASP_Identifier*		
Derivation Path : *[DerivationPath]*		
Comments : *[FreeText]*		
Parameter Name	**Parameter Value**	**Comments**
⋮ *ASP_ParIdOrMacro* ⋮	⋮ *ConstraintValue&Attributes* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*		

Proforma 38 – ASP Constraint Declaration

The Parameter Name and Parameter Value columns shall either be both present or both omitted.

This proforma is used for ASPs in the same way that the PDU Constraint Declaration proforma is used (see 13.4) except that encoding information is not relevant and shall not be specified.

SYNTAX DEFINITION:

555 ConsId&ParList ::= ConstraintIdentifier [FormalParList]

532 ASP_ConstraintGroupReference ::=[(SuiteIdentifier| TTCN_ModuleIdentifier)"/"]{ASP_ConstraintGroupIdentifier"/"}

349 ASP_Identifier ::=Identifier

558 DerivationPath ::= {ConstraintIdentifier Dot}+

357 ASP_ParIdOrMacro ::=ASP_ParId&FullId| MacroSymbol

562 ConstraintValue&Attributes ::=ConstraintValue ValueAttributes

13.4 PDU Constraint Declarations

In the tabular format a constraint is defined by specifying a value and optional attributes for each PDU field. The following information shall be supplied for each PDU constraint:

a) The name of the constraint,

   which may be followed by an optional formal parameter list.

b) The PDU type name.

c) The derivation path (see 13.6).

d) The encoding rules to be used for the Constraint.

   In order to specify explicit encodings for entire PDU Constraints, which override the encoding rules applicable to the given PDU type, this optional entry shall reference an entry in the relevant Encoding Definitions table (e.g. to change from BER to DER). If this entry is not used, then the encoding rules applicable to the PDU type apply. See 11.16.4.

e) The Encoding Variations to be used for the Constraint.

In order to specify explicit Encoding Variations for entire PDU Constraints, which override the Encoding Variations applicable to the given PDU type, this optional entry shall reference an entry in the relevant Encoding Variations table [e.g. to change from SD to LD(3)]. If this entry is not used, then the Encoding Variations applicable to the PDU type apply. See 11.16.4.

f) A constraint value for each field,

where the following information shall be supplied for each field:

1) Its name.

Each field entry in the field name column shall have been declared in the relevant PDU type definition. If any of the original PDU fields is defined as having both a short name and full identifier, the constraint shall not repeat the full identifier.

If the PDU definition refers to a Structured Type by macro expansion (i.e., with "<-" in place of the PDU field name) then in a corresponding constraint either:

– the individual elements from the Structured Type shall be included directly within the constraints; or

– the macro symbol (<-) shall be placed in the corresponding position in the PDU field name column of the constraint and the value shall be a reference to a constraint for the Structured Type referenced from the PDU definition.

Use of structured constraints by macro expansion in a constraint shall not be used unless the corresponding PDU definition also references the same Structured Type by macro expansion.

2) Its value and an optional attribute.

3) Optionally, a specific encoding identifier followed by any necessary actual parameter list, in order to specify explicit encodings for individual fields of a PDU Constraint, which override the encoding rules and encoding variations applicable to the PDU Constraint as a whole, and which override any specific field encoding applicable to this field for PDUs of this PDU type; the encoding identifier, if any, shall identify either one of the Encoding Variations or an Invalid Field Encoding Definition defined in the test suite [e.g. LD(10)]; see 11.16.4.

The encoding mechanism shall not be used with ASP constraints.

This information shall be provided in the format shown in Proforma 39, below.

PDU Constraint Declaration			
Constraint Name : *ConsId&ParList*			
Group : *[PDU_ConstraintGroupReference]*			
PDU Type : *PDU_Identifier*			
Derivation Path : *[DerivationPath]*			
Encoding Rule Name : *[EncodingRuleIdentifier]*			
Encoding Variation : *[EncVariationCall]*			
Comments : *[FreeText]*			
Field Name	**Field Value**	**Field Encoding**	**Comments**
⋮	⋮	⋮	⋮
PDU_FieldIdOrMacro	*ConstraintValue&Attributes*	*[PDU_FieldEncodingCall]*	*[FreeText]*
⋮	⋮	⋮	⋮
Detailed Comments: *[FreeText]*			

Proforma 39 – PDU Constraint Declaration

The Field Name and Field Value columns shall either be both present or both omitted. The Field Encoding column shall not be present as a single column on its own.

SYNTAX DEFINITION:

555 ConsId&ParList ::= ConstraintIdentifier[FormalParList]
551 PDU_ConstraintGroupReference ::=[(SuiteIdentifier| TTCN_ModuleIdentifier)"/"]{PDU_ConstraintGroupIdentifier "/"}
383 PDU_Identifier ::= Identifier
558 DerivationPath ::={ConstraintIdentifier Dot}+
452 EncodingRuleIdentifier ::= Identifier
511 EncVariationCall ::= EncVariationIdentifier[ActualParList]
391 PDU_FieldOrMacro ::= PDU_Field&Id| MacroSymbol
562 ConstraintValue&Attributes ::= ConstraintValue ValueAttributes
515 PDU_FieldEncodingCall ::= EncVariationCal| InvalidFieldEncodingCall

EXAMPLE 52 – A constraint, called C1, on the PDU called PDU_A:

PDU Constraint Declaration		
Constraint Name : C1		
PDU Type : PDU_A		
Derivation Path :		
Comments :		
Field Name	**Field Value**	**Comments**
FIELD1 FIELD2 FIELD3	(4 .. INFINITY) TRUE "A STRING"	

13.5 Parameterization of constraints

Constraints may be parameterized using a formal parameter list. The actual parameters are passed to a constraint from a constraints reference in a behaviour description.

EXAMPLE 53 – A parameterized constraint:

PDU Constraint Declaration		
Constraint Name : C2(P1:INTEGER;P2:BOOLEAN)		
PDU Type : PDU_B		
Derivation Path :		
Comments :		
Field Name	**Field Value**	**Comments**
FIELD1 FIELD2 FIELD3	P1 P2 "A STRING"	
Detailed Comments: A possible reference to C2 from a Test Case or Test Step may be: C2(0,TRUE)		

13.6 Base constraints and modified constraints

For every ASP, PDU or CM type definition at least one base constraint may be specified. In the case in which an ASP or CM has no parameters or a PDU has no fields, constraints are irrelevant and hence base constraints are unnecessary. A base constraint specifies a set of base, or default, values or matching symbols for each and every field defined in the appropriate definition. There may be any number of base constraints for any particular PDU (see Appendix I for examples).

When a constraint is specified as a modification of a base constraint, any fields not re-specified in the modified constraint will default to the values or matching symbols specified in the base constraint. The name of the modified constraint shall be a unique identifier. The name of the base constraint which is to be modified shall be indicated in the derivation path entry in the constraint header. This entry shall be left blank for a base constraint. A modified constraint

can itself be modified. In such a case the Derivation Path indicates the concatenation of the names of the base and previously modified constraints, separated by dots (.) A dot shall follow the last modified constraint name. The rules for building a modified constraint from a base constraint are:

a) if a parameter or field and its corresponding value or matching symbol is not specified in the modified constraint, then the value or matching symbol in the parent constraint shall be used (i.e., the value is inherited);

b) if a parameter or field and its corresponding value or matching symbol is specified in the modified constraint, then the specified value or matching symbol replaces the one specified in the parent constraint.

13.7 Formal parameter lists in modified constraints

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

a) the modified constraint shall have the same parameter list as the base constraint; in particular, there shall be no parameters omitted from or added to this list;

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

c) parameterized ASP parameters or PDU in a base constraint fields shall not be modified or explicitly omitted in a modified constraint.

13.8 CM Constraint Declarations

The field values for CM constraints shall be provided in the format shown in Proforma 40, below.

CM Constraint Declaration		
Constraint Name : *ConsId&ParList*		
Group : *[CM_ConstraintGroupReference]*		
CM Type : *CM_Identifier*		
Derivation Path : *[DerivationPath]*		
Comments : *[FreeText]*		
Parameter Name	**Parameter Value**	**Comments**
⋮ *CM_ParIdOrMacro* ⋮	⋮ *ConstraintValue&Attributes* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*		

Proforma 40 – CM Constraint Declaration

The Parameter Name and Parameter Value columns shall either be both present or both omitted.

This proforma is used for CMs in the same way as the PDU Constraint Declaration proforma is used (see 13.4).

SYNTAX DEFINITION:

555 ConsId&ParList ::= ConstraintIdentifier[FormalParList]

605 CM_ConstraintGroupReference ::= [(SuiteIdentifier| TTCN_ModuleIdentifier) "/"]{CM_ConstraintGroupIdentifier "/"}

424 CM_Identifier ::= Identifier

558 DerivationPath ::={ConstraintIdentifier Dot}+

431 CM_ParIdOrMacro ::= CM_ParIdentifier| MacroSymbol

562 ConstraintValue&Attributes ::= ConstraintValue ValueAttributes

This proforma is used for CMs in the same way that the PDU Constraint Declaration proforma is used for PDUs.

14 Specification of constraints using ASN.1

14.1 Introduction

This clause describes a method of specifying Type, ASP and PDU constraints in ASN.1, in a way similar to the definition of tabular constraints. The normal ASN.1 value declaration is extended to allow the use of the matching mechanisms. Mechanisms to replace or omit parts of ASN.1 constraints, to be used in modified constraints, are also defined.

In other respects, ASN.1 is used in constraints in the same way that it is used in types. In particular:

a) For identifiers within an ASN.1 constraint the hyphen symbol ("-") shall not be used; the underscore symbol ("_") may be used instead.

b) ASN.1 constraints shall not use external value references as defined in Recommendations X.680.

c) ASN.1 comments can be used within the table body. The comments column shall not be present in this table. Comments in ASN.1 start with "--" and end with either the next occurrence of "--" or with "end of line", whichever comes first. This prevents a single ASN.1 comment from spanning several lines. "End of line" is not, however, a defined symbol in TTCN.MP. ATS specifiers are recommended to facilitate the exchange of ATSs in TTCN.MP by always closing ASN.1 comments with "--".

14.2 ASN.1 Type Constraint Declarations

Both ASN.1 ASP constraints and ASN.1 PDU constraints can be structured by using references to ASN.1 Test Suite Type constraints for values of complex fields. ASN.1 Test Suite Types are defined in the declarations part of the ATS.

The following information shall be supplied for each ASN.1 Type Constraint Declaration:

a) The name of the Constraint,

   which may be followed by an optional formal parameter list.

b) The ASN.1 Type name.

c) The derivation path (see 13.6 and 14.6).

   In order to specify explicit Encoding Variations for entire ASN.1 Type Constraints, which override both the Encoding Variations of the PDU Constraint that references this ASN.1 Type Constraint and the default global Encoding Variations for the test suite, this optional entry shall reference an entry in the relevant Encoding Variations table [e.g. to change from SD to LD(3)]; if this entry is not used, then the default Encoding Variations apply to all ASN.1 Type Constraints of this type, unless specifically overridden within a particular Constraint.

d) The Encoding Variations to be used for the Constraint.

   If an ASN.1 Constraint Declaration is a modification of an existing ASN.1 constraint, the name of the ASN.1 constraint that is taken as the basis of this modification shall be referenced in the table in the derivation path entry.

e) The constraint value,

   where the body of the ASN.1 Type Constraint table contains the ASN.1 Constraint Declaration with optional attributes; all constraint values and attributes defined in 12.6 can be used in ASN.1 constraints.

   In order to specify explicit encodings for individual values within an ASN.1 Type Constraint, which override all other Encoding Variations for the specific ASN.1 Type Constraint encodings [see c) above], the keyword **ENC** is used after the relevant value, followed by a specific encoding identifier and any necessary actual parameter list. The

encoding identifier shall identify either one of the Encoding Variations or an Invalid Field Encoding Definition defined in the test suite.

ASN.1 Type Constraint Declarations shall be specified in the format shown in Proforma 41, below.

ASN.1 Type Constraint Declaration	
Constraint Name	**:** *ConsId&ParList*
Group	**:** *[ASN1_TypeConstraintGroupReference]*
Structured Type	**:** *ASN1_TypeIdentifier*
Derivation Path	**:** *[DerivationPath]*
Encoding Variation	**:** *[EncVariationCall]*
Comments	**:** *[FreeText]*
Constraint Value	
ConstraintValue&AttributesOrReplace	
Detailed Comments: *[FreeText]*	

Proforma 41 – ASN.1 Type Constraint Declaration

SYNTAX DEFINITION:

555 ConsId&ParList ::= ConstraintIdentifier[FormalParList]

523 ASN1_TypeConstraintGroupReference
::=[(SuiteIdentifier| TTCN_ModuleIdentifier)"/"]{ASN1_TypeConstraintGroupIdentifier"/"}

116 ASN1_Identifier ::=Identifier

558 DerivationPath ::={ConstraintIdentifier Dot}+

511 EncVariationCall ::=EncVariationIdentifier[ActualParList]

595 ConstraintValue&AttributesOrReplace ::=ConstraintValue&Attribute| Replacement{Comma Replacement}

This proforma is used for ASN.1 Types in the same way that the ASN.1 PDU Constraint Declaration proforma is used (see 14.4).

14.3 ASN.1 ASP Constraint Declarations

The following information shall be supplied for each ASN.1 ASP Constraint Declaration:

a) The name of the constraint,

which may be followed by an optional formal parameter list.

b) The ASP type name.

c) The derivation path (see 13.6 and 14.6).

If an ASN.1 Constraint Declaration is a modification of an existing ASN.1 constraint, the name of the ASN.1 constraint that is taken as the basis of this modification shall be referenced in the table in the derivation path entry.

d) The constraint value,

where the body of the ASP constraint table contains the ASN.1 Constraint Declaration with optional attributes. All constraint values and attributes defined in 12.6 can be used in ASN.1 constraints.

ASN.1 ASP Constraint Declarations shall be specified in the format shown in Proforma 42, below.

ASN.1 ASP Constraint Declaration	
Constraint Name	**:** *ConsId&ParList*
Group	**:** *[ASN1ASP_ConstraintGroupReference]*
ASP Type	**:** *ASP_Identifier*
Derivation Path	**:** *[DerivationPath]*
Comments	**:** *[FreeText]*
Constraint Value	
ConstraintValue&AttributesOrReplace	
Detailed Comments: *[FreeText]*	

Proforma 42 – ASN.1 ASP Constraint Declaration

SYNTAX DEFINITION:

555 ConsId&ParList ::= ConstraintIdentifier[FormalParList]

542 ASN1_ASP_ConstraintGroupReference ::=[(SuiteIdentifier|
TTCN_ModuleIdentifier)"/"]{ASN1_ASP_ConstraintGroupIdentifier"/"}

349 ASP_Identifier ::=Identifier

558 DerivationPath ::={ConstraintIdentifier Dot}+

595 ConstraintValue&AttributesOrReplace ::=ConstraintValue&Attribute| Replacement{Comma Replacement}

This proforma is used for ASN.1 Types in the same way that the ASN.1 PDU Constraint Declaration proforma is used (see 14.4).

14.4 ASN.1 PDU Constraint Declarations

The following information shall be supplied for each ASN.1 PDU Constraint Declaration:

a) The name of the Constraint,

which may be followed by an optional formal parameter list.

b) The PDU type name.

c) The derivation path (see 13.6 and 14.6).

d) The encoding rules to be used for the Constraint.

In order to specify explicit encodings for entire ASN.1 PDU Constraints, which override the default global encoding rules for the test suite, this optional entry shall reference an entry in the relevant Encoding Definitions table (e.g. to change from BER to DER); if this entry is not used, then the default encoding rules apply to all ASN.1 PDU Type Constraints of this type, unless specifically overridden in a particular Constraint.

e) The Encoding Variations to be used for the Constraint.

In order to specify explicit Encoding Variations for entire ASN.1 PDU Constraints, which override the default global Encoding Variations for the test suite. This optional entry shall reference an entry in the relevant Encoding Variations table [e.g. to change from SD to LD(3)]; if this entry is not used, then the default Encoding Variations apply to all ASN.1 PDU Type Constraints of this type, unless specifically overridden in a particular Constraint.

If an ASN.1 Constraint Declaration is a modification of an existing ASN.1 constraint, the name of the ASN.1 constraint that is taken as the basis of this modification shall be referenced in the table in the derivation path entry.

f) The constraint value,

where the body of the PDU constraint table contains the ASN.1 Constraint Declaration with optional attributes; all constraint values and attributes defined in 12.6 can be used in ASN.1 constraints.

In order to specify explicit encodings for individual values within an ASN.1 PDU Constraint, which override the default global encoding rules or the specific ASN.1 PDU Constraint encodings [see c) and d) above], the keyword **ENC** is used after the relevant value, followed by a specific encoding identifier and any necessary actual parameter list. The encoding identifier shall identify either one of the Encoding Variations or an Invalid Field Encoding Definition defined in the test suite.

PDU Constraint Declarations shall be specified in the format shown in Proforma 43, below.

ASN.1 PDU Constraint Declaration	
Constraint Name	**:** *ConsId&ParList*
Group	**:** *[ASN1PDU_ConstraintGroupReference]*
PDU Type	**:** *PDU_Identifier*
Derivation Path	**:** *[DerivationPath]*
Encoding Rule Name	**:** *[EncodingRuleIdentifier]*
Encoding Variation	**:** *[EncVariationCall]*
Comments	**:** *[FreeText]*
Constraint Value	
ConstraintValue&AttributesOrReplace	
Detailed Comments: *[FreeText]*	

Proforma 43 – ASN.1 PDU Constraint Declaration

SYNTAX DEFINITION:

555 ConsId&ParList ::= ConstraintIdentifier[FormalParList]

592 ASN1_PDU_ConstraintGroupReference ::=[(SuiteIdentifier| TTCN_ModuleIdentifier)"/"]{ASN1_PDU_ConstraintGroupIdentifier"/"}

383 PDU_Identifier ::=Identifier

558 DerivationPath ::={ConstraintIdentifier Dot}+

452 EncodingRuleIdentifier ::= Identifier

511 EncVariationCall ::= EncVariationIdentifier[ActualParList]

595 ConstraintValue&AttributesOrReplace ::=ConstraintValue&Attribute| Replacement{Comma Replacement}

14.5 Parameterized ASN.1 constraints

ASN.1 constraints may be parameterized (see 13.5).

14.6 Modified ASN.1 constraints

ASN.1 constraints can be specified by modifying an existing ASN.1 constraint. Portions of a constraint can be respecified to create a new constraint by using the REPLACE/OMIT mechanism.

Particular parameters or fields of a base or a modified constraint may be identified through a list of field selectors in order to replace their defined value by a new value, or to omit the defined value. A ReferenceList consists of the field selector identifiers (defined in the corresponding type definition) separated by dots which uniquely identify a particular (possibly structured) field within a PDU (or ASP). First level fields can be identified by a single selector, whereas nested fields require the full path.

Replace values shall be used only when a derivation path is specified. Full ASN.1 values shall be used only when a derivation path is not specified. Values that are REPLACEd or OMITted may be structured.

SYNTAX DEFINITION:

596 Replacement ::= **REPLACE** ReferenceList **BY** ConstraintValue&Attributes| **OMIT** ReferenceList

597 ReferenceList ::= (ArrayRef| ComponentIdentifier| ComponentPosition) {ComponentReference}

If a field belongs to a SEQUENCE, SET or CHOICE structure, the position of the field in parentheses may be used as a replacement for the field selector identifier. This technique shall be used where the identifier is not provided in the declaration of the field.

14.7 Formal parameter lists in modified ASN.1 constraints

The requirements of 13.7 also apply to modified ASN.1 constraints.

14.8 ASP Parameter and PDU field names within ASN.1 constraints

When specifying a constraint for an ASP or PDU in ASN.1, the parameter or field identifiers defined in the ASN.1 type definition for SEQUENCE, SET and CHOICE types may be used in order to identify the particular ASP or PDU parameters or fields a value stands for. In the case of CHOICE types, the identifiers identifying the variant shall be used. For SEQUENCE types, parameter or field identifiers shall be used whenever the value definition becomes ambiguous because of omitted values for OPTIONAL parameters or fields. For SET types, parameter or field identifiers shall be used in all cases.

EXAMPLE 54 – Field values in an ASN.1 PDU constraint:

Assume the type definition:

ASN.1 PDU Type Definition	
PDU Name	**:** XY_PDU
PCO Type	**:**
Comments	**:**
Type Definition	
SET { field_1 INTEGER OPTIONAL, field_2 BOOLEAN, field_3 INTEGER OPTIONAL, field_4 INTEGER OPTIONAL }	

Then a possible constraint is:

ASN.1 PDU Constraint Declaration	
Constraint Name	**:** CONS1
PDU Type	**:** XY_PDU
Derivation Path	**:**
Comments	**:**
Constraint Value	
{ field_1 5, field_2 TRUE, field_3 3 } *-- field_4 is not specified=>omitted when sending* *-- if identifier field_3 was not used it would be ambiguous whether 3 was the value of field_3 of* *-- field_4, since both are OPTIONAL.*	

14.9 ASN.1 CM Constraint Declarations

The parameter values for CM constraints shall be provided in the format shown in Proforma 44, below.

ASN.1 CM Constraint Declaration	
Constraint Name	**:** *ConsId&ParList*
Group	**:** *[ASN1CM_ConstraintGroupReference]*
CM Type	**:** *CM_Identifier*
Derivation Path	**:** *[DerivationPath]*
Comments	**:** *[FreeText]*
Constraint Value	
ConstraintValue&AttributesOrReplace	
Detailed Comments:	*[FreeText]*

Proforma 44 – ASN.1 CM Constraint Declaration

SYNTAX DEFINITION:

555 ConsId&ParList ::= ConstraintIdentifier[FormalParList]

615 ASN1_CM_ConstraintGroupReference ::=[(SuiteIdentifier| TTCN_ModuleIdentifier)"/"]{ASN1_CM_ConstraintGroupIdentifier"/"}

424 CM_Identifier ::=Identifier

558 DerivationPath ::={ConstraintIdentifier Dot}+

595 ConstraintValue&AttributesOrReplace ::=ConstraintValue&Attribute| Replacement{Comma Replacement}

This proforma is used for CMs in the same way that the PDU Constraint Declaration proforma is used for PDUs.

15 Dynamic Part

15.1 Introduction

The Dynamic Part contains the main body of the test suite: the Test Case, the Test Step and the Default behaviour descriptions.

15.2 Test Case dynamic behaviour

15.2.1 Specification of the Test Case Dynamic Behaviour table

15.2.1.1 The title of the table shall be "Test Case Dynamic Behaviour".

15.2.1.2 The header shall contain the following information:

a) Test Case name,

giving a unique identifier for the Test Case described in the table;

b) Test Group Reference,

giving the full name of the lowest level to the group that contains the Test Case; that full name shall conform to the requirements of 9.2, and end with a slash (/);

c) Test Purpose,

an informal statement of the purpose of the Test Case, as given in the relevant test suite structure and test purposes Recommendation (if any) or equivalent part of the test suite Recommendation (if any);

d) Default Reference,

an identifier (including an actual parameter list if necessary) of a Default behaviour description, if any, which applies to the Test Case behaviour description (see 15.4).

15.2.1.3 The body of the table shall display the following columns and corresponding information:

a) an (optional) line number column (see 15.2.5),

which, if present, shall be placed at the extreme left of the table.

b) a label column,

where labels can be placed to identify the TTCN statements to allow jumps using the GOTO construct (see 15.14);

c) a behaviour description,

which describes the behaviour of the LT and/or UT in terms of TTCN statements and their parameters, using the tree notation (see 15.6);

d) a constraints reference column,

where constraint references are placed to associate TTCN statements in a behaviour tree with a reference to specific ASP and/or PDU values defined in the constraints part (see clause 12);

e) a verdict column,

where verdict or result information is placed in association with TTCN statements in the behaviour tree (see 15.17);

f) an (optional) comments column,

this column is used to place comments that ease understanding of TTCN statements by providing short remarks or references to additional text in the optional detailed comments section.

The columns c), d), e) and f) shall be displayed in that order, from left to right. It is recommended that the mandatory label column be placed at the left of the behaviour description. Alternately, the label column may be placed to the right of the behaviour description.

15.2.1.4 An (optional) footer can contain detailed comments.

15.2.2 Test Case Dynamic Behaviour proforma

The Test Case dynamic behaviour shall be provided in the format shown in Proforma 45, below.

Test Case Dynamic Behaviour					
Test Case Name	**:** *TestCaseIdentifier*				
Group	**:** *TestGroupReference*				
Purpose	**:** *FreeText*				
Configuration	**:** *TCompConfigIdentifier*				
Defaults	**:** *[DefaultRefList]*				
Comments	**:** *[FreeText]*				
No.	**Label**	**Behaviour Description**	**Constraint Ref**	**Verdict**	**Comments**
1	.	.	.	.	.
2	.	.	.	.	.
.	*[Label]*	*StatementLine*	*[ConstraintReference]*	*[Verdict]*	*[FreeText]*
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
.	.	*TreeHeader*	.	.	.
.	.	*StatementLine*	.	.	.
.	.	.	.	.	.
n	.	.	.	.	.
Detailed Comments: *[FreeText]*					

Proforma 45 – Test Case Dynamic Behaviour

The alternative position of the label column is shown in dotted lines.

Column headers of this proforma can be abbreviated to: **L, Cref, V** and **C**. This enables the behaviour tree column to be as wide as possible in cases of physical paper size limitations.

SYNTAX DEFINITION:

624 TestCaseIdentifier ::= Identifier

626 TestGroupReference ::= [SuiteIdentifier "/"]{TestGroupIdentifier "/"}

329 TcompConfigIdentifier ::= Identifier

630 DefaultRefList ::=DefaultReference{Comma DefaultReference}

667 Label ::=Identifier

679 StatementLine ::= (Event[Qualifier][AssignmentList][TimerOps])| (Qualifier[AssignmentList][TimerOps])| (AssignmentList[TimerOps])| TimerOps| Construct| ImplicitSend

657 TreeHeader ::= TreeIdentifier [FormalParList]

669 ConstraintReference ::=ConsRef| FormalParIdentifier| AnyValue

674 Verdict ::=Pass| Fail| Inconclusive| Result

15.2.3 Structure of the Test Case behaviour

Each Test Case contains a precise description of sequences of (anticipated) events and related verdicts. This description is structured as a tree, with TTCN statements as nodes in that tree and verdict assignments at its leaves. In many cases it is more efficient to use Test Steps as a means of substructuring this tree (see Figure 7).

In TTCN this explicit modularization is expressed using Test Steps and the ATTACH construct.

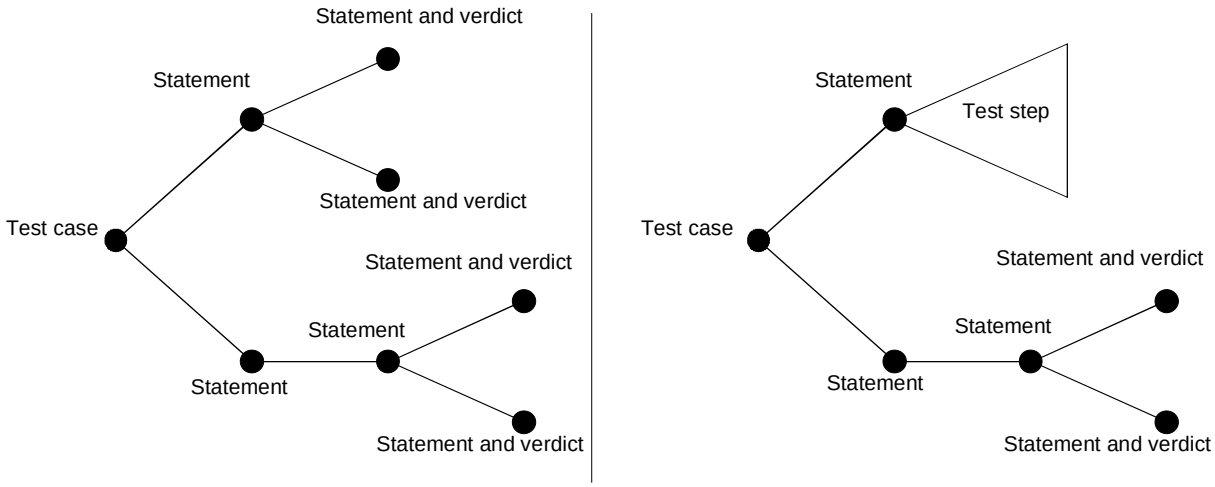


Figure 7/X.292 – Test Case Behaviour Structure

15.2.4 Concurrent Test Case Behaviour Description

If PTCs are used in a test case, then the header shall contain the additional entry, Configuration, which shall identify a Test Component Configuration declared in the Declaration Part.

The behaviour of the MTC is described by the first tree in the Test Case Behaviour table plus all attached trees. The MTC behaviour tree creates PTCs when required and associates each PTC with its own behaviour tree.

If a PTC behaviour is specified as a local tree in the test case behaviour, then the Defaults Reference shall be empty. This restriction prevents a PTC from inheriting the Default Behaviour of the MTC.

A test case shall only use the Test Components that are present in the referenced Test Component Configuration. The chosen configuration shall determine the set of PCOs and CPs that may be used in the test case. When used, the Configuration entry in the Test Case Dynamic Behaviour Header shall be provided in the format shown in Proforma 45.

15.2.5 Line numbering and continuation

Since lines in the behaviour description, when printed, may be too long to fit on one line, it is necessary to use additional symbols to indicate the extent of a single behaviour line. There are two available techniques:

a) Indicate the beginning of a new behaviour line; an extra line column is added as the leftmost column in the body of the table; there shall only be an entry in this column on those lines where a new behaviour line starts; the line numbers used shall be 1, 2, 3, ... and the numbering shall not be restarted when local trees are defined, i.e., there is a unique line number for each behaviour line of the behaviour table.

   NOTE 1 – The line numbers can be used for logging purposes, to record unambiguously which behaviour line was executed.

   NOTE 2 – The line numbers can be used as references in the detailed comments section.

b) Indicate the continuation of lines; if a line is to be continued within the behaviour description column a hash symbol (#) shall be placed in the leftmost position of the behaviour column, on the line of the continued text; it is recommended that the text of the continued part adopts the same level of indentation as the line it is continuing.

If a line is continued in any column other than the behaviour description column the hash symbol is not required.

EXAMPLE 55 – Printing long behaviour line:

55.1 Recommended style:

No.	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		This is a TTCN statement that is too long to print on a single line because the column is too narrow.	Ref1		
2		This is the next statement line	This is a constraint reference that is too long to print on one line		
3		An alternative statement line	Ref2		

55.2 Alternative style:

Label	Behaviour Description	Constraints Ref	Verdict	Comments
	This is a TTCN statement that is too long to print on a single line because the column is too narrow.	Ref1		
	This is the next statement line	This is a constraint reference that is too long to print on one line		
	An alternative statement line	Ref2		

15.3 Test Step dynamic behaviour

15.3.1 Specification of the Test Step Dynamic Behaviour table

The dynamic behaviour of Test Steps is defined using the same mechanisms as for Test Cases, except that Test Steps can be parameterized (see 15.7). Test Step dynamic behaviour tables are identical to Test Case dynamic behaviour tables, except for the following differences:

a) the table has the title "Test Step Dynamic Behaviour";

b) the first item in the header is the Test Step name,

which is a unique identifier for the Test Step followed by an optional list of formal parameters, and their associated types. These parameters may be used to pass PCOs, constraints or other data objects into the root tree of the Test Step;

c) the second item in the header is the Test Step Group Reference,

which gives the full name to the lowest level of the Test Step Library group that contains the Test Step; that full name shall conform to the requirements of (see 9.3), and end with a slash (/);

d) the third item in the header is the Test Step Objective,

which is an informal statement of the objective of the Test Step.

15.3.2 Test Step Dynamic Behaviour proforma

The Test Step dynamic behaviour shall be provided in the format shown in Proforma 46, below.

Test Step Dynamic Behaviour					
Test Step Name	**:** *TestStepId&ParList*				
Group	**:** *TestStepGroupReference*				
Objective	**:** *FreeText*				
Defaults	**:** *[DefaultRefList]*				
Comments	**:** *[FreeText]*				
No.	**Label**	**Behaviour Description**	**Constraint Ref**	**Verdict**	**Comments**
1 *2* .	. . *[Label]*	. . *StatementLine*	. . *[ConstraintReference]*	. . *[Verdict]*	. . *[FreeText]*
. . . . . . *n*	. . . .	. . . *TreeHeader* *StatementLine* . .	. . . .	. . . .	. . . .
Detailed Comments: *[FreeText]*					

Proforma 46 – Test Step Dynamic Behaviour

The alternative position of the label column is shown in dotted lines.

Column headers of this proforma can be abbreviated to: **L, Cref, V** and **C**.

SYNTAX DEFINITION:

638 TestStepId ::= TestStepIdentifier[FormalParList]
641 TestStepReference ::= [SuiteIdentifier "/"]{TestStepIdentifier "/"}
630 DefaultRefList ::=DefaultReference{Comma DefaultReference}
667 Label ::=Identifier
679 StatementLine ::= (Event[Qualifier][AssignmentList][TimerOps])| (Qualifier[AssignmentList][TimerOps])| (AssignmentList[TimerOps])| TimerOps| Construct| ImplicitSend
657 TreeHeader ::=TreeIdentifier [FormalParList]
669 ConstraintReference ::=ConsRef| FormalParIdentifier| AnyValue
674 Verdict ::=Pass| Fail| Inconclusive| Result

15.4 Default dynamic behaviour

15.4.1 Default behaviour

A TTCN Test Case shall specify alternative behaviour for *every* possible event (including invalid ones). It often happens that in a behaviour tree every sequence of alternatives ends in the same behaviour. This behaviour may be factored out as default behaviour to this tree. Such Default behaviour descriptions are located in the global Default Library.

The dynamic behaviour of Defaults is defined using the same mechanisms as for Test Steps, except for the following restrictions:

a) it is not permitted to specify Default behaviour for the Default behaviour;

b) a default behaviour description may attach local trees (see 15.7.1) but shall not attach Test Steps;

c) if local trees are used in a Default behaviour description, they shall not attach Test Steps;

d) the tree(s) in the behaviour description shall not use the ACTIVATE operation (see 15.18.4).

Both PCOs and other actual parameters may be passed to Default behaviour descriptions in the same way that they may be passed to Test Steps. The same rules on scope and textual substitution of these parameters apply as described for tree attachment (see 15.13).

15.4.2 Specification of the Default Dynamic Behaviour table

Default dynamic behaviour tables are identical to Test Step dynamic behaviour tables, except for the following differences:

a) the table has the title "Default Dynamic Behaviour";

b) the first item in the header is the Default name,

which is a unique identifier for the Default followed by an optional list of formal parameters, and their associated types. These parameters may be used to pass PCOs, constraints or other data objects into the root tree of the Default;

c) the second item in the header is the Default Group Reference,

which gives the full name of the lowest level to the Default Group that contains the Default; that full name shall conform to the requirements of 9.4, and end with a slash (/);

d) the third item in the header is the Default Objective,

which is an informal statement of the objective of the Default.

15.4.3 Default Dynamic Behaviour proforma

The Default dynamic behaviour shall be provided in the format shown in Proforma 47, below.

Default Dynamic Behaviour					
Default Name : *DefaultId&ParList*					
Group : *DefaultGroupReference*					
Objective : *FreeText*					
Comments : *[FreeText]*					
No.	**Label**	**Behaviour Description**	**Constraint Ref**	**Verdict**	**Comments**
1	.	.	.	.	.
2	.	.	.	.	.
.	*[Label]*	*StatementLine*	*[ConstraintReference]*	*[Verdict]*	*[FreeText]*
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
.	.	*TreeHeader*	.	.	.
.	.	*StatementLine*	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
n	.	.	.	.	.
Detailed Comments: *[FreeText]*					

Proforma 47 – Default Dynamic Behaviour

The alternative position of the label column is shown in dotted lines.

Column headers of this proforma can be abbreviated to: **L, Cref, V** and **C**.

SYNTAX DEFINITION:

649 DefaultId&ParList ::=DefaultIdentifier{FormalParList}

651 DefaultGroupReference ::= [SuiteIdentifier "/"]{DefaultGroupIdentifier "/"}

667 Label ::=Identifier

679 StatementLine ::= (Event[Qualifier][AssignmentList][TimerOps])| (Qualifier[AssignmentList][TimerOps])| (AssignmentList[TimerOps])| TimerOps| Construct| ImplicitSend

657 TreeHeader ::= TreeIdentifier [FormalParList]

669 ConstraintReference ::=ConsRef| FormalParIdentifier| AnyValue

674 Verdict ::=Pass| Fail| Inconclusive| Result

15.5 Behaviour description

The behaviour description column of a dynamic behaviour table contains the specification of the combinations of TTCN statements that are deemed possible by the test suite specifier. The set of these combinations is called the behaviour tree. Each TTCN statement is a node in the behaviour tree.

15.6 Tree notation

Each TTCN statement shall be shown on a separate statement line. The statements can be related to one another in two ways:

– as sequences of TTCN statements;

– as alternative TTCN statements.

Sequences of TTCN statements are represented one statement line after the other, each new TTCN statement being indented once from left to right, with respect to its predecessor.

EXAMPLE 56 – TTCN statements in sequence:

```
EVENT_A
    CONSTRUCT_B
        EVENT_C
```

Statements at the same level of indentation and belonging to the same predecessor node represent the possible alternative statements which may occur at that time. Henceforth, this set of TTCN statements will be referred to as the *set of alternatives*, or simply *alternatives*.

EXAMPLE 57 – Alternative TTCN statements:

```
CONSTRUCT_A1
STATEMENT_A2
EVENT_A3
```

EXAMPLE 58 – Combining sequences and alternatives to build a tree:

```
EVENT_A
    CONSTRUCT_B
        EVENT_C
    STATEMENT_D1
    EVENT_D2
```

Whether a TTCN statement can be evaluated successfully or not depends on various conditions associated with the statement line. These conditions are not necessarily mutually exclusive, i.e. it is possible that for any given moment more than one statement line could be evaluated successfully. Since statement lines are evaluated in the order of their appearance in the set of alternatives, the first statement with a fulfilled condition will be successful. This might lead to unreachable behaviour; in particular if statements are encoded as alternatives, following statements that are always successful.

REPEAT and GOTO are always successful. In addition, SEND, IMPLICIT SEND, assignments and timer operations are successful provided that the accompanying qualifier, if any, evaluates to TRUE.

Graphical indentation of statement lines in the TTCN.GR form is mapped to indentation values in TTCN.MP. Statements in the first level of alternatives having no predecessor in the root or local tree they belong to, shall have the indentation value of zero. Statements having a predecessor shall have the indentation value of the predecessor plus one as their indentation value.

SYNTAX DEFINITION:

664 Line ::= **$Line** Indentation StatementLine

EXAMPLE 59 – $Line [6] +R1_POSTAMBLE

15.7 Tree names and parameter lists

15.7.1 Introduction

Each behaviour description shall contain at least one behaviour tree. In order that trees may be unambiguously referred to (such as in an ATTACH construct) each tree has a tree name.

The first tree appearing within a behaviour description is called the root tree. The name of a root tree is the identifier appearing in the header of its dynamic behaviour table. That is, the tree name of the root tree of a Test Step is the Test Step Identifier for that Test Step, and likewise for root trees in Test Case dynamic behaviours and Default dynamic behaviours.

Trees other than the root tree which appear within dynamic behaviour tables are termed local trees. Local trees are prefixed by a tree header which contains the tree name.

SYNTAX DEFINITION:

654 RootTree ::={BehaviourLine}+

655 LocalTree ::=Header {BehaviourLine}+

15.7.2 Trees with parameters

All trees, except Test Case root trees, may be parameterized. The parameters may provide PCOs, constraints, variables, or other such items for use within the tree. Test Case root trees shall not be parameterized.

If a tree is parameterized, then a list of formal parameters and their types shall appear within parentheses directly following the tree name. For example, the formal parameter list for a Test Step root tree shall appear within parentheses immediately following the Test Step Identifier in the header of the Test Step dynamic behaviour table. Similarly, the formal parameter list for a local tree shall appear immediately after the tree name in the tree header.

In constructing the formal parameter list, each formal parameter shall be followed by a colon and the name of the type of the formal parameter. If more than one formal parameter of the same type is present, these may be combined into a sublist. When such a sublist is used, the formal parameters within the sublist shall be separated from each other by a comma. The final formal parameter in the sublist shall be followed by a colon and the formal parameter type.

When there is more than one formal parameter and type pair (or more than one sublist and type pair), the pairs shall be separated from each other by semi-colons.

Formal parameters may be of PCO type, ASP type, PDU type, structure type or one of the other predefined or Test Suite Types.

If a formal parameter of a tree is type **PDU,** then specific fields in the PDU shall not be referenced in the tree. If the formal parameter is a specific PDU identifier, then specific fields in the PDU may be referenced in the tree.

EXAMPLE 60 – A Test Step using formal parameters: EXAMPLE_TREE (L:TSAP; X:INTEGER; Y:INTEGER)

EXAMPLE 61 – A Test Step using a formal parameters with a sublist: EXAMPLE_TREE (L:TSAP; X, Y:INTEGER)

15.8 TTCN statements

The tree notation allows the specification of test events initiated by the Lower Tester(s) or Upper Tester(s) (SEND and IMPLICIT SEND events), test events received by the Lower Tester(s) or Upper Tester(s) (RECEIVE, OTHERWISE, TIMEOUT and DONE), constructs (GOTO, ATTACH, REPEAT, CREATE, RETURN and ACTIVATE) and pseudo-events comprising combinations of qualifiers, assignments and timer operations. These are collectively known as TTCN statements.

Test events can be accompanied by qualifiers (Boolean expressions), assignments and timer operations. Qualifiers, assignments and timer operations can also stand alone, in which case they are called pseudo-events.

15.9 TTCN test events

15.9.1 Sending and receiving events

TTCN supports the initiation (sending) of ASPs and PDUs to named PCOs and acceptance (receipt) of ASPs and PDUs at named PCOs. The PCO model is defined in 11.10 and 15.9.5.3.Concurrent TTCN supports the sending and receiving of CMs to named CPs. The CP model is defined in 11.11.

SYNTAX DEFINITION:

682 Send ::=[PCO_Identifier| CP_Identifier| FormalParIdentifier] "!"(ASP_Identifier| PDU_Identifier| CM_Identifier)

684 Receive ::= [PCO_Identifier| CP_Identifier| FormalParIdentifier] "?" (ASP_Identifier| PDU_Identifier| CM_Identifier)

In the simplest form, an ASP identifier or PDU identifier follows the SEND symbol (!) for events to be initiated by the LT or UT, or a RECEIVE symbol (?) for events which it is possible for the LT or UT to accept. The optional PCO name is not provided. This form is valid when there is only one PCO in the test suite.

EXAMPLE 62 – !CONreq or ?CONind

If more than one PCO exists in a test suite, then a PCO name appearing in the declarations part, or in the formal parameter list of the tree, shall prefix the SEND symbol or the RECEIVE symbol. The PCO name is used to indicate the PCO at which the test event may occur.

EXAMPLE 63 – L! CONreq or L? CONind

In the case of CPs, the CP identifier shall be used and shall prefix the SEND symbol in the case of sending a CM and shall prefix the RECEIVE symbol in the case of receiving a CM.

EXAMPLE 64 – A_CP!A_CM or A_CP?A_CM

15.9.2 Receiving events

A RECEIVE event line evaluates successfully if an incoming ASP or PDU on the specified PCO matches the event line. A match occurs if the following conditions are fulfilled:

a) the incoming PDU can be decoded in accordance with the applicable encoding rules;

b) the incoming ASP or PDU is valid according to the ASP or PDU type definition referred to by the event name on the event line. In particular, all parameters and/or field values shall be of the type defined, and satisfy any length restrictions specified;

c) the ASP or PDU matches the constraint reference on the event line;

d) in cases where a qualifier is specified on the event line, the qualifier shall evaluate to TRUE; the qualifier may contain references to ASP parameters and/or PDU fields.

The incoming event is removed from the PCO queue only when it successfully matches a RECEIVE event line.

In concurrent TTCN the receipt and matching of a CM at a CP is treated in the same manner as described above.

15.9.3 Sending events

A SEND event line with a qualifier is successful if the expression in the qualifier evaluates to TRUE. Unqualified SEND events are always successful. The outgoing ASP or PDU that results from a SEND event shall be constructed as follows:

a) All ASP parameter and PDU field values shall be of the type specified in the corresponding definitions, and will satisfy any length restrictions in the definitions.

b) The value of the ASP parameter and PDU fields shall be set as specified in the constraint referenced on the event line (see clauses 12, 13 and 14 for an explanation of constructing ASPs or PDUs with constraints).

c) Any direct assignments to ASP parameters or PDU fields on the event line will supersede the corresponding value specified in the constraint, if any.

d) All parameters and/or fields in the outgoing ASP or PDU shall contain specific values or be explicitly omitted prior to completion of the SEND event.

e) The fully constructed PDU shall be encoded in accordance with the applicable encoding rules.

Generation of an ASP parameter or PDU field value by either the constraints or assignments that violates the declared type and length restrictions shall cause a test case error.

In concurrent TTCN the sending of CMs at CPs is treated in the same manner as described above.

15.9.4 Lifetime of events

Identifiers of ASP parameters and PDU fields associated with SEND and RECEIVE shall be used only to reference ASP parameter and PDU field values on the statement line itself.

In the case of SEND events, relevant ASP parameters and PDU fields can be set, if required, in appropriate assignments on the SEND line.

EXAMPLE 65 – !A_PDU (A_PDU.FIELD:=3)

The effects of such an assignment shall not persist after the event line in which they occurred.

In the case of RECEIVE events, if relevant ASP parameter and PDU field values need to be subsequently referenced, either the whole ASP or PDU or a relevant part of it shall be assigned to variables on the RECEIVE line itself. These variables may then be referenced in subsequent lines.

EXAMPLE 66 – ?A_PDU (VAR:=A_PDU.FIELD)

where VAR may be used on event lines subsequent to receipt of A_PDU.

The lifetime of CMs is also restricted to the relevant RECEIVE statement. Identifiers of CM fields may be accessed in a similar manner as identifiers of PDU fields.

EXAMPLE 67 – A_CP!A_CM or A_CP?A_CM

15.9.5 Execution of the behaviour tree

15.9.5.1 Introduction

The test suite specifier shall organize the behaviour tree representing a Test Case or a Test Step according to the following rules regarding test execution:

a) Starting from the root of the tree, the LT or UT remains on the first level of indentation until an event matches. If an event is to be initiated the LT or UT initiates it; if an event is to be received, it is said to match only if a received real event occurs and matches the event line.

b) Once an event has matched, the LT or UT moves to the next level of indentation. No return to a previous level of indentation can be made, except by using the GOTO construct.

c) Event lines at the same level of indentation and following the same predecessor event line represent the possible alternatives which may match at that time. Alternatives shall be given in the order that the test suite specifier requires the LT or UT to attempt either to initiate or receive them, if necessary, repeatedly, until one matches.

EXAMPLE 68 – Illustration of a TTCN behaviour tree:

Suppose that the following sequence of events can occur during a test whose purpose is to establish a connection, exchange some data, and close the connection. The events occur at the lower tester PCO L:

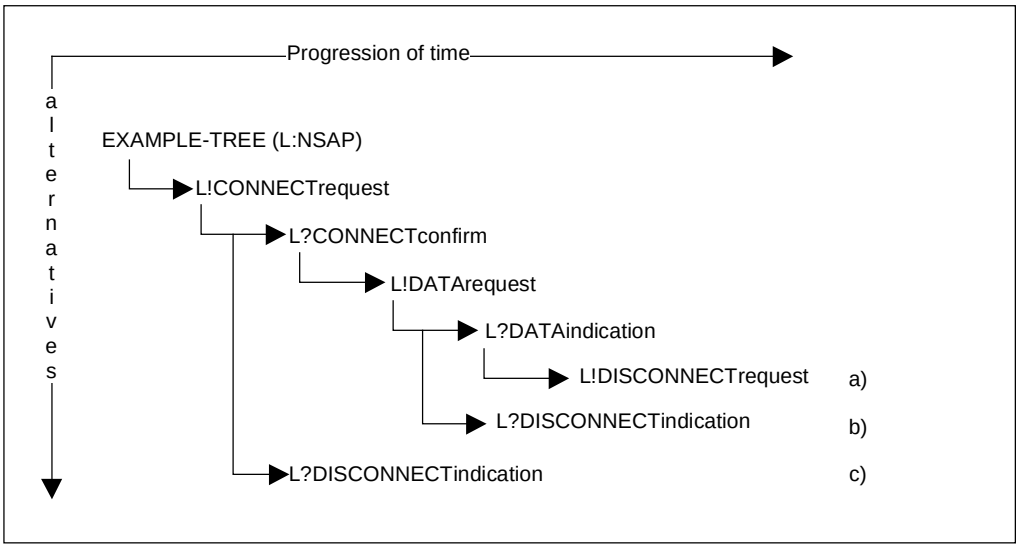
a) CONNECTrequest, CONNECTconfirm, DATArequest, DATAindication, DISCONNECTrequest.

Progress can be thwarted at any time by the IUT or the service-provider. This generates two more sequences:

b) CONNECTrequest, CONNECTconfirm, DATArequest, DISCONNECTindication.

c) CONNECTrequest, DISCONNECTindication.

The three sequences of events can be expressed as a TTCN behaviour tree. There are five levels of alternatives, and only three leaves [a) to c)], because the SEND events L! are always successful. Execution is to progress from left to right (sequence), and from top to bottom (alternatives). The following figure illustrates this progression, and the principle of the TTCN behaviour tree:



T0731090-98\d08

There are no lines, arrows or leaf names in TTCN. The behaviour tree of the previous example would be represented as follows:

EXAMPLE 69 – A TTCN behaviour tree:

Test Step Dynamic Behaviour					
Test Step Name	**:** TREE_EX_1(L:NSAP)				
Group	**:** TTCN_EXAMPLES/TREE_EXAMPLE_1/				
Objective	**:** To illustrate the use of trees.				
Default	**:** NOTE – This example can be simplified by using Defaults.				
No.	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		L!CONNECTrequest	CR1		Request …
2		L?CONNECTconfirm	CC1		…Confirm
3		L!DATArequest	DTR1		Send Data
4		L?DATAindication	DTI1		Receive Data
5		L!DISCONNECTrequest	DSCR1	PASS	Accept
6		L?DISCONNECTindication	DSCI1	INCONC	Premature
7		L?DISCONNECTindication	DSCR1	INCONC	Premature

15.9.5.2 Concept of snapshot semantics

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

15.9.5.3 Restrictions on using events

In order to avoid test case errors, the following restrictions apply:

a) A Test Case or Test Step should not contain behaviour where the relative processing speed of the Means of Testing (MOT) could impact the results. To prevent such problems, a RECEIVE, OTHERWISE or TIMEOUT event line shall only be followed by other RECEIVE, OTHERWISE and TIMEOUT event lines in a set of alternatives. As a consequence, Default trees shall contain only RECEIVE, OTHERWISE and TIMEOUT event lines on the first set of alternatives.

b) Once there is an event on a PCO or CP queue or a timeout in the timeout list, it can be removed from the queue or list only by a successful match of the related TTCN statement. In the case of a set of alternatives that includes RECEIVE statements. The set of expected incoming events shall be fully specified. This means that it shall be a test case error if, during execution, no match of any of the RECEIVE statements occurs and yet execution progresses to the next level of alternatives because of a TIMEOUT which occurred after an ASP or PDU, that was not specified in the set of RECEIVE statements, was received on any one of the relevant PCO or CP queues. IMPLICIT SEND shall not be used with CMs.

c) Precautions should be taken when using concurrent TTCN to avoid unreliable results caused by situations in which the order of receipt of events at different PCOs or CPs is used to determine verdict assignment. The actual time at which PDU or CM is received, relative to the receipt of other PDUs or CMs, may not be accurately reflected when executing parallel test components.

EXAMPLE 70 – An incomplete set of RECEIVE events:

PARTIAL_TREE	
!A START T	
?B	PASS
?TIMEOUT T	
!C	PASS
?D	

a)

PARTIAL_TREE	
!A START T	
?B	PASS
?OTHERWISE	FAIL
?TIMEOUT T	
!C	
?D	PASS

b)

In a) if D is received in response to !A, the test case will assign an erroneous PASS verdict by virtue of the TIMEOUT. This can be avoided by using the OTHERWISE statement.

d) In concurrent TTCN, the relative ordering of events at different PCOs or different CPs should not affect the verdict assigned, since this would lead to unrepeatability of results caused by differences in processing and transmission speeds.

15.9.5.4 Precautions when using concurrent TTCN

Precautions should be taken when using concurrent TTCN to avoid unrepeatable results caused by situations in which the order of receipt of events at different PCOs or at different CPs is used to determine verdict assignment. The actual time at which a PDU or CM is received, relative to the receipt of other PDUs or CMs, may not be accurately reflected when executing parallel test components.

15.9.6 IMPLICIT SEND event

In the Remote Test Methods, although there is no explicit PCO above the IUT, it is necessary to have a means of specifying, at a given point in the description of the behaviour of the LT, that the IUT should be made to initiate a particular PDU or ASP (but not CM). For this purpose, the implicit send event is defined, with the following syntax:

SYNTAX DEFINITION:

683 ImplicitSend ::= "<" IUT "!" (ASP_Identifier | PDU_Identifier)">"

The **IUT** in the syntax takes the place of the PCO identifier used with a normal SEND or RECEIVE, indicating that the specified ASP or PDU is to be sent by the IUT. The angle brackets signify that this is an implicit event, i.e., there is no specification of what is done to the IUT to trigger this reaction, only a specification of the required reaction itself.

An IMPLICIT SEND event is always considered to be successful, in the sense that any alternatives coded after, and at the same level of indentation as the IMPLICIT SEND are unreachable.

An IMPLICIT SEND shall be used only where the relevant OSI Recommendation(s) permit the IUT to send the specified ASP or PDU at that point in its communication with the LT.

For every IMPLICIT SEND in a test suite, the test suite specifier shall create and reference a question in the partial PIXIT proforma that permits indication of whether the IMPLICIT SEND can be invoked on demand.

An IMPLICIT SEND event shall not be used unless the test method being used is one of the Remote Test Methods. An IMPLICIT SEND event shall not be used unless the same effect could have been achieved using the DS test method.

NOTE – For example, when testing a connection-oriented Transport Protocol implementation, if this restriction did not exist it would be permissible to use IMPLICIT SEND to get the IUT to initiate a CR TPDU because in the DS test method that effect could be achieved by getting the UT to send a T-CONreq ASP. On the other hand, it would not be permissible to use IMPLICIT SEND to get the IUT to initiate an N-RstReq ASP because that effect could not be controlled through the Transport Service boundary. The reason for this restriction is to prevent Test Cases from requiring greater external control over an IUT than is provided for in the relevant protocol Recommendation.

When an IMPLICIT SEND event is specified, the associated internal events within the IUT necessary to meet the requirements of the Recommendation for the protocol being tested are also performed, e.g. set timer, initialize state variables.

The semantics of IMPLICIT SEND is that the SUT shall be controlled as necessary in order to cause the initiation of the specified ASP or PDU. The way in which the SUT is to be controlled should be specified in the PIXIT (or documentation referenced by the PIXIT).

Neither a final verdict nor a preliminary result shall be associated with an IMPLICIT SEND event.

At an appropriate point following an IMPLICIT SEND, there should be a RECEIVE event to match the ASP or PDU that should, as a result, have been sent by the IUT.

EXAMPLE 71 – EXAMPLE use of IMPLICIT SEND:

Test Case Dynamic Behaviour					
Test Case Name	**:**	IMPI			
Group	**:**	TTCN_EXAMPLES/			
Purpose	**:**	A partial tree to illustrate the use of IMPLICIT SEND			
Default	**:**				
Comments	**:**				
No.	**Label**	**Behaviour Description**	**Constraint Ref**	**Verdict**	**Comments**
:		:			
5		<IUT!CR>	CR1		
6		L?CR	CR1		
7		L!CC	CC1		
:		:			
12		L?OTHERWISE			
:		:			

15.9.7 OTHERWISE event

The predefined event OTHERWISE is the TTCN mechanism for dealing with unforeseen test events in a controlled way. OTHERWISE has the syntax:

SYNTAX DEFINITION:

685 Otherwise ::= [PCO_Identifier| CP_Identifier| FormalParIdentifier] "?" **OTHERWISE**

OTHERWISE is used to denote that the LT or UT shall accept a*ny* incoming event which has not previously matched one of the alternatives to the OTHERWISE. The tester shall accept any incoming data that it has not been possible to decode or that has not matched a previous alternative to this OTHERWISE event.

In non-concurrent TTCN, if more than one PCO exists in a test suite, then either a PCO name appearing in the declarations part, or a formal parameter from the formal parameter list of the tree where that formal parameter is used to convey a PCO name, shall prefix the OTHERWISE. The PCO name is used to indicate the PCO at which the test event may occur. Incoming events, including OTHERWISE, are considered only in terms of the given PCO.

EXAMPLE 72 – Use of OTHERWISE with PCO identifiers:

PARTIAL_TREE	
PCO1?A	
PCO2?B	PASS
PCO1?C	INCONC
PCO2?OTHERWISE	FAIL

Assume no event is received at PCO1, then receipt of event B at PCO2 results in a PASS verdict. Receipt of any other event at PCO2 results in a FAIL verdict.

Due to the significance of ordering of alternatives, incoming events which are alternatives following an unconditional OTHERWISE on the same PCO will never match.

EXAMPLE 73 – Incoming events following an OTHERWISE:

PARTIAL_TREE	
PCO1?A	PASS
PCO1?OTHERWISE	FAIL
PCO1?C	INCONC

The OTHERWISE will match any incoming event other than A. The last alternative, ?C, can never be matched.

15.9.8 OTHERWISE and concurrent TTCN

In concurrent TTCN, OTHERWISE may be used with CPs as well as PCOs. OTHERWISE on CPs is allowed to provide an efficient way of handling "all other CMs on this CP".

15.9.9 TIMEOUT event

The TIMEOUT event allows expiration of a timer, or of all timers, to be checked in a Test Case. When a timer expires (conceptually immediately before a snapshot processing of a set of alternative events), a TIMEOUT event is placed into a timeout list. The timer becomes immediately inactive. Only one entry for any particular timer may appear in the list at any one time. Since TIMEOUT is not associated with a PCO, a single timeout list is used.

When a TIMEOUT event is processed, if a timer name is indicated, the timeout list is searched, and if there is a timeout event matching the timer name, that event is removed from the list, and the TIMEOUT event succeeds.

If no timer name is indicated, any TIMEOUT event in the timeout list matches. The TIMEOUT event succeeds if the list is not empty. When this occurs, the entire timeout list is immediately emptied.

TIMEOUT has the following syntax:

SYNTAX DEFINITION:

686 Timeout ::= "?"TIMEOUT [TimerIdentifier| FormalParIdentifier]

EXAMPLE 74 – Use of TIMEOUT:

	?TIMEOUT T			

Since TIMEOUT events are not RECEIVE events they are not rendered unreachable by previously listed OTHERWISE alternatives.

15.9.10 Concurrent TTCN events and constructs

The CREATE construct and the DONE event are used in concurrent TTCN.

15.9.10.1 CREATE construct

The Main Test Component is started at the beginning of Test Case execution. The Main Test Component starts Parallel Test Components, as needed, by means of the CREATE construct, which has the following syntax:

SYNTAX DEFINITION:

692 Create ::= CREATE "(" CreateList ")"

This construct invokes a set of Parallel Test Components. For each PTC, there are two arguments. The first is the identifier of the PTC that is created, which shall match the identifier of a PTC in the Test Component Configuration referenced in the test case header. The second is a reference to a behaviour tree (i.e. Test Step or local tree), possibly with a parameter list containing actual values (e.g. PCOs and CPs). The effect of the CREATE construct is that each PTC listed starts executing its behaviour description in parallel with the execution of the Main Test Component.

NOTE – Passing PCO and CP identifiers to a behaviour tree as actual parameters allows the same behaviour tree to be used in more than one test component.

The PCOs and CPs used in the execution of the behaviour description associated with a PTC by the CREATE construct shall only be those determined by the Test Component Configuration for that Test Case.

The execution of a CREATE construct on a PTC which has already been created shall result in a Test Case error. The execution of a CREATE by any Test Component other than the MTC shall result in a test case error.

In the CREATE construct, PCO identifiers and CP identifiers are passed to a PTC by textual substitution, as is usual in the ATTACHment of Test Steps. All others parameters are passed by value. This is done to prevent side effects on variables which could affect the processing of other PTCs, causing unrepeatable results.

15.9.10.2 DONE event

When the MTC terminates, the final verdict is assigned by the MTC, as calculated up to this moment (see 15.17.5). The DONE event can be used in the MTC and the PTCs to find out whether PTCs have already terminated. Test Components can use this information to determine their own preliminary results and further actions; in particular, the MTC can avoid terminating before all PTCs have terminated (see 15.17.5).

SYNTAX DEFINITION:

687 Done ::= "?" DONE "("[TcompIdList] ")"

A missing argument list is interpreted as being a list of all PTCs stated in a CREATE constructs executed prior to the execution of the DONE event. A DONE event without an argument list shall only be used by the MTC.

EXAMPLE 75 – Use of the DONE event:

```
PARTIAL_MTC_TREE

CREATE(PTC1:TREEA)
  CREATE(PTC2:TREEB)
      START T1
          ?DONE(PTC1,PTC2)
          ?TIMEOUT T1                FAIL
```

NOTE 1 – It is recommended to use ?TIMEOUT as an alternative to ?DONE.

NOTE 2 – If DONE is the only alternative, it amounts to an order to wait for the specified PTCs to terminate.

NOTE 3 – DONE is not a means for the MTC to coordinate termination of PTCs. Termination can only be achieved by providing an appropriate exchange of CMs. TTCN does not offer any predefined CMs for this purpose.

15.10 Expressions

15.10.1 Introduction

There are two kinds of expressions in TTCN: assignments and Boolean expressions. Both assignments and Boolean expressions may contain explicit values and the following forms of reference to data objects:

a) Test Suite Parameters;

b) Test Suite Constants;

c) Test suite and Test Case Variables;

d) Formal parameters of a Test Step, Default or local tree;

e) ASPs and PDUs (on event lines).

Any variables occurring in Boolean expressions and/or on the right hand side of an assignment shall be bound. If an unbound variable is used this is a test case error.

SYNTAX DEFINITION:

```
703  Expression ::= SimpleExpression {RelOpSimpleExpression}
704  SimpleExpression ::= Term {AddOp Term}
705  Term ::= Factor {MultiplyOp Factor}
706  Factor ::=[UnaryOp]Primary
707  Primary ::=Value| DataObjectReference| OpCall| SelectExprIdentifier| "("Expression ")"
739  Value ::=LiteralValue| ASN1_Value[ASN1_Encoding]
740  LiteralValue ::=Number| BooleanValue| Bstring| Hstring| Ostring| Cstring| R_Value
741  Number ::=(NonZeroNum {Num})| 0
742  NonZeroNum ::= 1| 2| 3| 4| 5| 6| 7| 8| 9
743  Num::=0|  NonZeroNum
744  BooleanValue ::= TRUE| FALSE
745  Bstring ::= """{Bin| Wildcard} """B
746  Bin ::= 0| 1
747  Hstring ::= """{Hex| Wildcard} """H
748  Hex ::= Num| A| B| C| D| E| F
749  Ostring ::= """{Oct| Wildcard} """O
750  Oct ::=Hex Hex
751  Cstring ::= """ {Char| Wildcard| "\"} """
752  Char ::= /*REFERENCE-A character defined by the relevant CharacterString type. */
753  Wildcard ::=AnyOne| AnyOrNone
754  AnyOne ::= "?"
755  AnyOrNone ::= "*"
708  DataObjectReference {ComponentReference}
709  DataObjectIdentifier ::=TS_ParIdentifier| TS_ConstIdentifier| TS_VarIdentifier| TC_VarIdentifier|
       FormalParIdentifier| ASP_Identifier| PDU_Identifier| CM_Identifier| VarIdentifier
710  ComponentReference ::= RecordRef| ArrayRef| BitRef
711  RecordRef ::= Dot(ComponentIdentifier| ComponentPosition)
712  ComponentIdentifier ::= ASP_ParIdentifier| PDU_FieldIdentifier| CM_ParIdentifier| ElemIdentifier| ASN1_Identifier
714  ComponentPosition ::= "("Number ")"
715  ArrayRef ::=Dot "[" ComponentNumber "]"
716  ComponentNumber ::= Expression
```

717 BitRef ::= Dot (BitIdentifier| "["BitNumber "]")
718 BitIdentifier ::= Identifier
719 BitNumber ::= Expression
720 OpCall ::= OpIdentifier (ActualParList | "("")")
721 OpIdentifier ::=TS_ OpIdentifier| TS_ProcIdentifier| PredefinedOpIdentifier
722 PredefinedOpIdentifier::=BIT_TO_INT| HEX_TO_INT| INT_TO_HEX| IS_CHOSEN| IS_ PRESENT| LENGTH_OF| NUMBER_OF_ELEMENTS
723 AddOp ::= "+"| "/"| OR
724 MultiplyOp ::= "*"| "/"| MOD| AND
725 UnaryOp ::= "+"| "-"| NOT
726 RelOp ::= "="|"<"| ">"|"<>"| ">="|"<="

15.10.2 References for ASN.1 defined data objects

15.10.2.1 Introduction

In order to permit references to components of data objects defined using ASN.1, TTCN provides three access mechanisms: record references, array references and bit references.

SYNTAX DEFINITION:

708 DataObjectReference ::= DataObjectIdentifier {ComponentReference}
710 ComponentReference ::= RecordRef | ArrayRef | BitRef
711 RecordRef ::= Dot(ComponentIdentifier| ComponentPosition)
715 ArrayRef ::=Dot "[" ComponentNumber "]"
717 BitRef ::= Dot (BitIdentifier| "["BitNumber "]")

15.10.2.2 Record references

A record reference may be used to reference a component of a data object of the type SEQUENCE, SET or CHOICE. A record reference is constructed using a dot notation, appending a dot and the name (component identifier) or number (component position) of the desired component to the data object identifier. The component identifier, if defined, should be used in preference to the component position. References to unnamed components are constructed by giving within parentheses the number which is the position of the component within the type definition. By definition, the implicit numbering of components starts with zero; hence the third component has position number 2.

Recommendations X.680 defines SET types having unordered components. This is relevant only if values of that type are encoded and sent over the underlying service-provider. TTCN therefore treats data objects of SET type in the same way as objects of SEQUENCE type, i.e. referring to the components with number *i* always means a reference to the *i*th field as declared in the type.

After an ASP or PDU or CM has been received, referring to the component with the index *i* will always return the same value. There is no change of order of the elements in a SET by any operation in TTCN.

SYNTAX DEFINITION:

711 RecordRef ::= Dot (ComponentIdentifier | ComponentPosition)
712 ComponentIdentifier ::= ASP_ParIdentifier | PDU_FieldIdentifier | CM_ParIdentifier | ElemIdentifier | ASN1_Identifier
714 ComponentPosition ::= "(" Number ")"

EXAMPLE 76 – Component record references:

```
Example_type ::= SEQUENCE {
                 field_1                    INTEGER
                 field_2     BOOLEAN,
                             OCTET STRING }
```

If var1 is of ASN.1 type Example_type, then the following could be written:

var1.field_1	which refers to the first (INTEGER) field
var1.(3)	which refers to the third (unnamed) field

EXAMPLE 77 – PDU field references:

```
Example_type ::= SEQUENCE {
                                :
                                user-data    OCTET STRING,
                                :    }
```

On a statement line that contains XY_PDUtype, the following could be written:

L? XY_PDU (buffer := XY_PDUtype.user_data)

in order to load the variable buffer with the contents of the user_data field of the incoming PDU.

When a PDU or an ASN.1 type parameter, field or element is chained to an ASP, another PDU, or a CM, a record reference may be used to identify a component of that PDU or ASN.1 type. The record reference shall identify the relevant complete sequence of parameter, field or element names separated by dots, starting with a data object identifier which resolves to the relevant ASP identifier, CM identifier, or (if ASPs are not used in the test suite) PDU identifier. Beyond this initial data object identifier the sequence shall not contain any PDU identifiers or ASN.1 type identifiers, but rather just the identifiers of the relevant parameters, fields and elements. This mechanism shall not be used if there is any ambiguity about the identity of a PDU constraint or ASN.1 type constraint in the sequence. The following example illustrates the use of record references when chaining of constraints is used (see 12.4).

EXAMPLE 78 – Record references with chaining:

```
ASN.1 ASP Type Definition
ASP1_type ::= SEQUENCE  {
                          par1 OCTET STRING,
                          par2 OCTET STRING,
                          Pdu1 PDU1_type
                        }
PDU1_type ::= SEQUENCE   {
                          Field1 OCTET STRING
                          Field2 OCTET STRING
                          F       F_type
                          }
                    ASN.1 Structure Type Definition
F_type ::= SEQUENCE  {
                       Data1 IA5String
                       Data2 IA5String
                     }
```

When using constraints of type ASP1_type, PDU1_type and F_type, the values of data1 and data2 may be referenced as follows:

ASP1_Type.pdu1.F.data1

ASP1_Type.pdu1.F.data2

Similarly the whole PDU field F may be referenced as:

ASP1_Type.pdu1.F

or the whole PDU may be referenced as:

ASP1_Type.pdu1

It should be noted that the declarations used in this example could apply to both static chaining and dynamic chaining, as the differences between the two types of chaining are only visible in the constraints. Thus, the record reference is independent of the variety of chaining used.

15.10.2.3 Array references

An array reference may be used to reference a component of a data object of the type SEQUENCE OF or SET OF. An array reference shall be constructed using a dot notation, appending a dot and the index of the desired component to the data object identifier. The index, giving the position of the component within the data object (when the object is viewed as a linear array), is enclosed within square brackets. By definition within ASN.1, the indexing of components starts with zero. The index may be an expression, in which case it shall evaluate to a non-negative INTEGER.

Recommendations X.680 defines SET OF types having unordered components. This is relevant only if values of that type are encoded and sent over the underlying service-provider. TTCN therefore treats data objects of SET OF type in the same way as objects of SEQUENCE OF type, i.e., referring to the components with number *i* always means a reference to the *i*th field as declared in the type.

After an ASP or PDU or CM has been received, referring to the component with the index *i* will always return the same value. There is no change of order of the elements in a SET OF by any operation in TTCN.

SYNTAX DEFINITION:

715 ArrayRef ::= Dot "[" ComponentNumber "]"

716 ComponentNumber ::= Expression

EXAMPLE 79 – Component array references:

```
Array_type ::= SEQUENCE OF {BOOLEAN}
```

If var2 is of ASN.1 type Array_type, then the following could be written in order to refer to the first BOOLEAN in the sequence:

var2.[0]

var1.[1-1]

15.10.2.4 Bit references

A bit reference may be used to reference particular bits within a BITSTRING type. For this purpose, data objects of BITSTRING type are assumed to be defined as SEQUENCE OF {BOOLEAN}. Thus, a bit reference may be constructed using the index notation as for array references. The leftmost bit has the index zero. An expression used as an index in a bit reference shall evaluate to a non-negative INTEGER. Alternatively, if certain bits of a BITSTRING are associated with an identifier (named bits), then this identifier may be used to refer to the bit.

SYNTAX DEFINITION:

717 BitRef ::= Dot (BitIdentifier | "[" BitNumber "]")

718 BitIdentifier ::= Identifier

719 BitNumber ::= Expression

EXAMPLE 80 – Bit references:

```
B_type ::= BITSTRING {ack(0),poll(3)}
```

This defines a BITSTRING type B_type where bit zero is called "ack" and bit three is called "poll".

If b_str is of ASN.1 type B_type, then the following could be written:

b_str.ack := TRUE

b_str.[2] := FALSE

Note that b_str.poll := TRUE and b_str.[3] := TRUE both assign the value TRUE to the "poll" bit.

15.10.3 References for data objects defined using tables

The same syntax as defined in 15.10.2.2 shall be used to construct record references to components of ASPs, PDUs, CMs and Structured Types defined in tabular form. Chaining of ASPs, PDUs, CMs and Structured Types in tabular form affects record references in exactly the same way as it does for those defined in ASN.1.

Where a parameter, field or element is defined to include an item which is a true substructure of a type defined in a Structured Type table, a reference to the item in the substructure shall consist of the record reference to the parameter, field or element followed by a dot and the identifier of the item within that Structure.

Where a Structure is used as a macro expansion, the elements in the Structure shall be referenced to as if it was expanded into the Structure referring to it.

If a parameter, field or element is defined to be of meta-type **PDU** no reference shall be made to fields of that substructure.

15.10.4 Assignments

15.10.4.1 Introduction

Test events may be associated with a list of assignments and/or a qualifier. Assignments are separated by commas and the list is enclosed in parentheses.

SYNTAX DEFINITION:

701 AssignmentList ::= "(" Assignment {Comma Assignment } ")"

702 Assignment ::=DataObjectReference ":=" Expression

During execution of an assignment the right-hand side shall evaluate to an element of the type of the left-hand side.

The effect of an assignment is to bind the Test Case or Test Suite Variable (or ASP parameter or PDU field) 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 81 – Use of assignments with event lines:

```
(X:=1)
  (Y:=2)
    L!A (Y:=0, X:=Y, A.field1:=y)
      L?B (Y:=B.field2, X:=X+1)
```

When PDU A is successfully transmitted the contents of the Test Case Variables X and Y will be zero, and field1 of PDU A will also contain zero. Upon receipt of PDU B the Test Case Variable Y would be assigned the contents of field2 from PDU B and the Test Case Variable X would be incremented.

15.10.4.2 Assignment rules for string types

If length-restricted string types are used within an assignment, the following rules apply:

a) if the destination string type is defined to be shorter than the source string, the source string is truncated on the right to the maximum length of the destination string type;

b) if the source string is shorter than that allowed by the destination string type, then the source string is left-aligned and padded with fill characters up to the maximum size of the destination string type.

Fill characters are:

– " " (blank) for all CharacterStrings;

– "0" (zero) for BITSTRINGs, HEXSTRINGs and OCTETSTRINGs.

When an unbounded (i.e., arbitrary length) string type variable is used on the left-hand side of an assignment, it shall become bound to the value of the right-hand side without padding. Padding is only necessary when the variable is of a fixed length string type.

15.10.5 Qualifiers

An event may be qualified by placing a Boolean expression enclosed in square brackets after the event. This qualification shall be taken to mean that the statement is executed only if both the event matches and the qualifier evaluates to TRUE.

If both a qualifier and an assignment are associated with the same event, then the qualifier shall appear first, any term in it being evaluated with the values holding before execution of the assignment.

SYNTAX DEFINITION:

681 Qualifier :: = "[" Expression "]"

15.10.6 Event lines with assignments and qualifiers

An event may be associated with an assignment, a qualifier or both. If an event is associated with an assignment, the assignment is executed only if the event matches. If an event is associated with a qualifier, the event may match only if the qualifier evaluates to TRUE. If an event is associated with both, the event may match only if the qualifier evaluates to TRUE, and the assignment is executed only if the event matches.

If a RECEIVE event is qualified and the event that has occurred potentially matches the specified event, then the qualifier shall be evaluated in the context of the event that has occurred. If the qualifier contains a reference to ASP parameters and/or PDU fields, then the values of those parameters and/or fields are taken from the event that has occurred.

The rules for use of assignments within events are as follows:

a) on a SEND event all assignments are performed *after* the qualifier is evaluated and *before* the ASP or PDU is transmitted;

b) on SEND events assignments are allowed for the fields of the ASP or PDU being transmitted;

c) on a RECEIVE event assignments are performed after the event occurs and cannot be made to fields of the ASP or PDU just received.

An assignment to a constraint ASP parameter, PDU field or structure element in the behaviour part will overwrite constraint values on a SEND event line.

EXAMPLE 82 – Use of a qualified SEND event:

```
PARTIAL_TREE
!A[X=3]
!B
```

Processing these alternative SEND events, the tester will send A only if the value of the variable X is 3. Otherwise it will send B.

The OTHERWISE event may be used together with qualifiers and/or assignments. If a qualifier is used, this Boolean becomes an additional condition for accepting any incoming event. If an assignment statement is used, the assignment will take place only if all conditions for matching the OTHERWISE are satisfied.

EXAMPLE 83 – Using OTHERWISE, qualifiers and assignments:

```
PARTIAL_TREE(PCO:XSAP; PCO2; YSAP)
PCO1?A                                                    PASS
PCO2?B                                                    INCONC
PCO1?C                                                    PASS
PCO2?OTHERWISE [X <> 2] (Reason:= "X not equal 2")        FAIL
PCO2?OTHERWISE (Reason:= "X equals 2 but event not B")    FAIL
```

Assume that no event is received at PCO1. Receipt of event B at PCO2 when X = 2 gives an inconclusive verdict. Receipt of any other event at PCO2 when X <> 2 results in a FAIL verdict and assigns a value of "X not equal 2" to the CharacterString variable Reason. If an event is received at PCO2 that satisfies neither of these scenarios, then the final OTHERWISE will match.

Events involving CMs occurring at CPs may also be associated with an assignment, a qualifier or both, in the same manner as for PDUs, as described above.

EXAMPLE 84 – CMs associated with a qualifier:

```
A_CP!A_CM                [X=2]
```

15.11 Pseudo-events

It is permitted to use assignments, qualifiers and timer operations by themselves on a statement line in a behaviour tree, without any associated event. These stand-alone expressions are called pseudo-events.

The meaning of such a pseudo-event is as follows:

a) If only a qualifier is specified: The qualifier is evaluated and execution continues with subsequent behaviour, if the qualifier evaluates to TRUE; if it evaluates to FALSE the next alternative is attempted. If no alternative exists, then this is a test case error.

b) If only assignments and/or timer operations are specified: The assignments shall be executed from left to right and/or the timer operations shall be executed from left to right.

c) If assignments and/or timer operations are specified preceded by a qualifier: The qualifier shall be evaluated first and the assignments and/or timer operations shall be evaluated only if the qualifier evaluates to TRUE.

15.12 Timer management

15.12.1 Introduction

A set of operations is used to model timer management. These operations can appear in combination with events or as stand-alone pseudo-events.

Timer operations can be applied to:

– an individual timer, which is specified by following the timer operation by the timer name;

– all timers, which is specified by omitting the timer name.

It is assumed that the timers used in a test suite are either inactive or running**.** All running timers are automatically cancelled at the end of each Test Case. There are three predefined timer operations: START, CANCEL and READTIMER. More than one timer operation may be specified on a event line if necessary. This is indicated by separating the operations by commas.

When a timer operation appears on the same statement line as an event and/or a qualifier, the timer operation shall be executed if and only if the event matches and/or the qualifier evaluates to TRUE.

SYNTAX DEFINITION:

727 TimerOps ::=TimerOp {CommaTimerOp}

728 TimerOps ::=StartTimer | CancelTimer | ReadTimer

15.12.2 START operation

The START operation is used to indicate that a timer should start running.

SYNTAX DEFINITION:

729 StartTimer ::= START (TimerIdentifier)["("TimerValue ")"]

301 TimerIdentifier ::= Identifier

731 TimerValue ::= Expression

The optional timer value parameter shall be used if no default duration is given, or if it is desired to assign an expiry time (i.e., duration) for a timer that overrides the default value specified in the timer declarations.

Timer values shall be of type INTEGER. The test case writer shall ensure that the optional timer value parameter shall evaluate to a positive non-zero INTEGER. A test case error shall result if a timer is started with a zero or negative value.

Any variables occurring in the expression specifying the optional timer value shall be bound. If an unbound variable is used this is a test case error.

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 will use the duration stated in the timer declarations part.

EXAMPLE 85 – Uses of START timer:

The Ti are timer identifiers and the Vi are timer values:
START T0
START T0 (V0)
START T1, START T2 (V2)

The START operation may be applied to a running timer, in which case the timer is cancelled, reset and started. Any entry in the timeout list for this timer shall be removed from the timeout list.

15.12.3 CANCEL operation

The CANCEL operation is used to stop a running timer.

SYNTAX DEFINITION:

730 CancelTimer ::= CANCEL [TimerIdentifier| FormalParIdentifier]
301 TimerIdentifier::=Identifier
731 TimerValue ::=Expression

A cancelled timer becomes inactive. If a TIMEOUT event for that timer is in the timeout list, that event is removed from the timeout list. If the timer name on the CANCEL operation is omitted, all running timers become inactive and the timeout list is emptied.

Canceling an inactive timer is a valid operation, although it does not have any effect.

EXAMPLE 86 – Some uses of CANCEL timer:

where the Ti are timer identifiers:
CANCEL
CANCEL T0
CANCEL T1, CANCEL T2
CANCEL T1, START T3

15.12.4 READTIMER operation

The READTIMER operation is used to retrieve the time that has passed since the specified timer was started and to store it into the specified Test Suite or Test Case Variable. This variable shall be of type INTEGER. The time value assigned to the variable is interpreted as having the time unit specified for the timer in its declaration. By convention, applying the READTIMER operation on an inactive timer will return the value zero.

SYNTAX DEFINITION:

732 ReadTimer ::= READTIMER (TimerIdentifier| FormalParIdentifier) "("DataObjectReference")"
301 TimerIdentifier ::= Identifier

EXAMPLE 87 – Using READTIMER:

```
:
START TimerName(TimeVal)
      ?EVENT_A
            +Tree_A
      ?EVENT_B
            +Tree_B
      ?EVENT_C
            READTIMER TimerNAME(CurrTime)
      ?TIMEROUT TimerName
:
```

If EVENT_C is received prior to expiration of the timer named by TimerName, the amount of time which has passed since starting the timer will be stored in the Test Case or Test Suite Variable CurrTime. The behaviour contained in Tree_C may use the value of this Test Suite or Test Case Variable.

EXAMPLE 88 – READTIMER used in combination with other timer operations:

```
READTIMER T1 (PASSED_TIME), CANCEL T1
READTIMER T1 (V1), START NEW_TIMER (V1)
```

15.13 ATTACH construct

15.13.1 Introduction

Trees may be attached to other trees by using the ATTACH construct, which has the syntax:

SYNTAX DEFINITION:

696 Attach ::= "+" TreeReference [ActualParList]
698 TreeReference ::= TestStepIdentifier | TreeIdentifier
699 ActualParList ::= "(" ActualPar{Comma ActualPar} ")"
700 ActualPar ::= Value | PCO_Identifier | CP_Identifier | TimerIdentifier

Test suite and Test Case Variables are global to both the tree that does the attachment (the main tree) and the attached tree, i.e., any changes made to variables in an attached tree also apply to the main tree. Tree attachment constructs shall appear on a statement line by themselves.

15.13.2 Scope of tree attachment

Behaviour descriptions may contain more than one tree. However, only the *first* tree in the behaviour description is accessible from outside the behaviour description. Any subsequent trees are considered to be Test Steps local to the behaviour description, and thus not externally accessible.

It should be noted that only Test Cases are directly executable, while Test Steps are executed only if attached to a Test Case, or to a Test Step whose point of attachment can be traced back to a Test Case (either directly or via other attached Test Steps). Test Cases are not attachable.

Tree reference may be Test Step Identifiers or tree identifiers, where:

a) a Test Step Identifier denotes the attachment of a Test Step that resides in the Test Step Library; the Test Step is referenced by its unique identifier;

b) a tree identifier shall be the name of one of the trees in the current behaviour description; this is attachment of a local tree.

15.13.3 Tree attachment basics

Given a behaviour tree, it is possible to detach parts of this tree in the form of separate behaviour trees, i.e., Test Steps. The points where a Test Step has been cut out of the original tree are indicated by the attach symbol (+) followed by the name assigned to the Test Step.

EXAMPLE 89 – Partitioning a large tree into two smaller trees:

```
TOP_TREE
A
    A1
    D1
        D11
        D12
C
    C1
        D1
```

is equivalent to:

```
TOP_TREE
A
    A1
    +STEP
C
    C1
C


        +STEP
```

and

```
STEP
D
    D11
    D12
```

This operation can be performed not only on the main behaviour tree of the Test Case (the root tree) but also on the Test Steps detached from it. The attached tree will either be a local tree or a member of the Test Step Library.

Tree attachment can be defined in a more general way than the mere re-insertion of complete Test Steps:

– An attached tree need not contain full paths down to the leaves of the tree it is attached to (its *calling tree*). Rather, some subsequent behaviour common to all paths of the attached tree may be specified in the calling tree, namely as behaviour subsequent to the attachment line.

– Some (even top level) lines of the attached Test Step may again have the form +SOME_SUBTREE, calling for the attachment of further Test Steps.

– Attached Test Steps may be parameterized.

15.13.4 Meaning of tree attachment

15.13.4.1 The following list defines the tree attachment execution semantics:

a) The attachment line (e.g., +STEP) in the behaviour tree (e.g., TOP_TREE) is formally one (e.g., A_i) in an ordered set of alternatives:

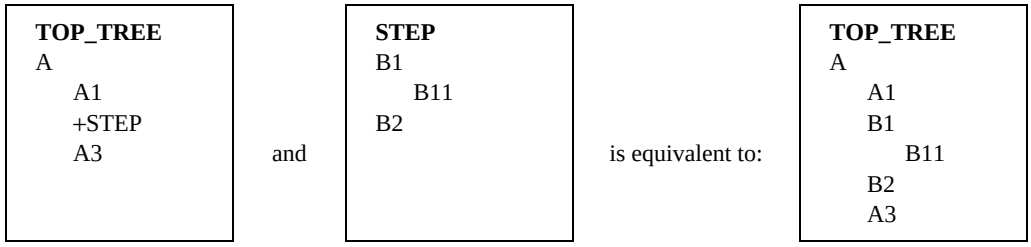
$$(A_1, ..., A_i, ..., A_n)$$

Attaching STEP in this position means expanding the TOP_TREE by inserting the Test Step STEP's top alternatives, e.g., $(B_1, ... , B_m)$ into this sequence, yielding a new sequence:

$$(A_1, ..., A_{(i-1)}, B_1, ..., B_m, A_{(i+1),}, ..., A_n)$$

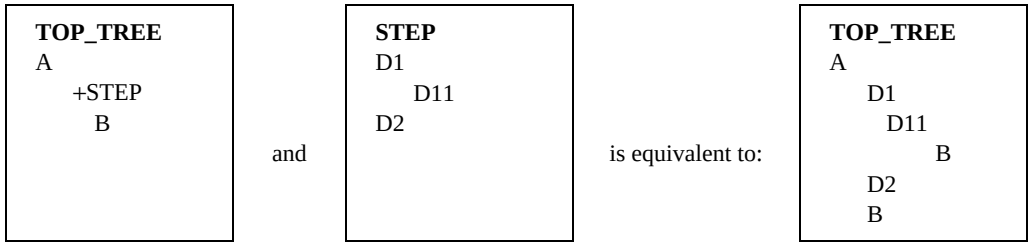
of alternatives. Any subsequent behaviour to the Bs will be attached together with them.

EXAMPLE 90 – Expansion of a Test Step:



b) Any behaviour subsequent to the +STEP line in the tree will become behaviour subsequent to all the leaves of the attached STEP expanded into the tree.

EXAMPLE 91 – Subsequent behaviour to an ATTACH:



c) When an actual parameter list is used on an ATTACH construct, then the actual parameter shall be substituted for each corresponding formal parameter using simple textual substitution. This substitution shall take place according to the following scoping rules:

1) actual parameters on the ATTACH of a local tree shall be substituted for corresponding formals only directly within that local tree;

2) actual parameters on the ATTACH of a root tree of a Test Step are substituted for all occurrences of the corresponding formals within the root tree and any local trees directly within the Test Step;

3) when a parameterized tree is attached:

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

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

iii) formal and actual parameters of test steps shall be used in such a way that only valid TTCN is created by textual substitution.

EXAMPLE 92 – Substitution of parameters:

```
TOP_TREE(L:NSAP;U:TSAP)
L!CONreq(M:=1)
    +STEP(U,M,2)
```

and

```
STEP (PCO:TSAP;X,Y:INTEGER)
PCO?CONind(X:Y)
```

is equivalent to:

```
TOP_TREE(L:NSAP;U:TSAP)
L!CONreq(M:=1)
    U?CONind(M:=2)
```

EXAMPLE 93 – Scoping rules for parameter substitution:

Test Step Dynamic Behaviour

Test Case Name : TEST_STEP_1(X,Y:INTEGER)
Group : TTCN_EXAMPLES/PARAMS/STEPS/
Object : To illustrate scoping rules for substitution
Default :
Comments :

No.	Label	Behaviour Description	Constraint Ref	Verdict	Comments
1		?A	A1		
2		+TEST_STEP_2(X)			
3		+LOCAL(5)			
		LOCAL(F:INTEGER)			
4		!B	B1		
5		(TC_VAR:=F+Y)		PASS	

Detailed Comments:

When TEST_STEP1 is attached by a calling tree, all occurrences of the formal parameters X and Y within the entire Test Step (including within the local tree LOCAL) will be replaced with the actuals provided. Note that formals X and Y are not automatically substituted with actuals within TEST_STEP2. However, the actual parameter value for formal X is substituted in the ATTACH construct "+TEST_STEP2(X)". This results in the substitution of the actual parameter value X (in TEST_STEP1) for whatever formal parameter appears in the declaration of TEST_STEP2. Finally, note that actual parameter (constant) 5 is substituted for formal "F" when the tree LOCAL is attached. This substitution takes place only within the local tree.

15.13.5 Passing parameterized constraints

Constraints may be passed as parameters to Test Steps. If the constraint has a formal parameter list, then the constraint shall be passed together with an actual parameter list. The actual parameters of the constraint shall already be bound at the point of attachment.

EXAMPLE 94 – Passing a parameterized constraint:

Suppose that the constraint C1 has a single formal parameter of type INTEGER. TOP_TREE attaches STEP and passes C1 as a parameter. Note that the constraints reference in STEP is not parameterized:

```
TOP_TREE
    :
    +STEP(C1(3))
    :
```

```
STEP(PARA:A_PDU)
    :
    !A_PDU              PAR
    :
```

15.13.6 Recursive tree attachment

As tree attachment works recursively (STEP may contain a +SOME_OTHER_TREE line) the tree expansion semantics may never lead to a tree free of attachment lines.

EXAMPLE 95 – A legal recursive tree attachment:

```
TOP_TREE
A
+STEP
    B
```

and

```
STEP
C
        D
    +TOP_TREE
```

one expansion is equivalent to:

```
TOP_TREE
A
C
    D
        B
+TOP_TREE
  B
```

A tree shall not attach itself, either directly or indirectly, at its top level of indentation.

NOTE – It is unnecessary to expand either any Test Step that will not be executed, or any alternatives beyond the current level until an alternative from the current level has been selected.

EXAMPLE 96 – An illegal recursive tree attachment:

```
TOP_TREE
A
    +STEP
      B
```

and

```
STEP
C
  +TOP_TREE
D
```

one expansion is equivalent to:

```
TOP_TREE
A
 C
+TOP_TREE
  B
 C
  B
```

15.13.7 Tree attachment and Defaults

The expansion of Defaults in a tree shall be completed before this tree is attached anywhere (see 15.18.5).

NOTE – Special care has to be taken where both tree attachment and Defaults are used in a behaviour description.

15.14 Labels and the GOTO construct

A label may be placed in the labels column on any statement line in the behaviour tree.

NOTE 1 – Whenever an entry is executed in the behaviour tree for which a label is specified, that label should be recorded in the conformance log in such a way that it can be associated with the record of the execution of that entry.

A GOTO to a label may be specified within a behaviour tree provided that the label is associated with the first of a set of alternatives, one of which is an ancestor node of the point from which the GOTO is to be made. A GOTO shall be used only for jumps within one tree, i.e., within a Test Case root tree, a Test Step tree, a Default tree or a local tree. As a consequence, each label used in a GOTO construct shall be found within the same tree in which the GOTO is used. No GOTO shall be made to the first level of alternatives of local trees, Test Steps or Defaults.

A GOTO shall not refer to a label prior to an ACTIVATE construct which is an ancestor node of the GOTO.

A GOTO shall be specified by placing an arrow (→) or the keyword GOTO, followed by the name of the label, on a statement line of its own in the behaviour tree.

SYNTAX DEFINITION:

695 GoTo ::= ("→"| GOTO) Label

A label shall be unique within a tree. If a GOTO is executed, the Test Case shall proceed with the set of alternatives referred to by the label.

GOTOs shall always be unconditional and therefore always execute.

NOTE 2 – A Boolean expression may be placed as the immediate ancestor of a GOTO to gain the effect of a conditional jump.

EXAMPLE 97 – Use of GOTO:

Test Case Dynamic Behaviour					
Test Case Name	**:** GOTO_EX1				
Group	**:** TTCN_EXAMPLES/GOTO_EXAMPLE1/				
Object	**:** To illustrate use of labels and GOTO				
Default	**:**				
Comments	**:**				
No.	**Label**	**Behaviour Description**	**Constraint Ref**	**Verdict**	**Comments**
1	LA	?A	A1		
2	LB	?B	B1		
3	LB2	+B-tree			
4	LC	?C	C1		
5	LD	[D=1]			
6		→LA			
7	LE	[E=1]			
8	LF	!F	F1	FAIL	
Detailed Comments: This example shows a jump to LA. From the same position in that tree it would also be allowed to jump to LB or LD, but it would not be allowed to jump to LB2 or LF (because the set of alternatives does not contain an ancestor node of the point from which the jump is made) nor to LC or LE (because these are not the first of a set of alternatives).					

15.15 REPEAT construct

This subclause describes a mechanism to be used in behaviour descriptions for iterating a Test Step a number of times. The syntax of this REPEAT construct is:

SYNTAX DEFINITION:

697 Repeat **::= REPEAT** TreeReference [ActualParList]UNTIL Qualifier

The tree reference shall be a reference to either a local tree or a Test Step defined in the Test Step Library. For the rules of attachment see 15.13. The REPEAT construct has the following meaning: first the tree, referred to by the tree reference, is executed. Then, the qualifier is evaluated. If the qualifier evaluates to TRUE, execution of the REPEAT construct is completed. If not, the tree is executed again, followed by evaluation of the qualifier. This process is repeated until the qualifier evaluates to TRUE.

The REPEAT construct can always be executed and should be the last alternative of a series of TTCN statements at the same level of indentation, as allowed by 15.9.5.3 a).

NOTE – The REPEAT construct is recommended, if applicable, instead of use of GOTO.

EXAMPLE 98 – Use of REPEAT (see also Appendix I):

Test Case Dynamic Behaviour

Test Case Name	**:**	RPT_EX1
Group	**:**	TTCN_EXAMPLES/REPEAT_EXAMPLE1/
Object	**:**	To illustrate use of REPEAT
Default	**:**	
Comments	**:**	

No.	Label	Behaviour Description	Constraint Ref	Verdict	Comments
1		(FLAG:=FALSE)			
2		!A	A1		
3		REPEAT STEP1 (FLAG)UNTIL(FLAG)			
4		!D	D1	PASS	
		STEP1 (F:BOOLEAN)			
		?B(F:=TRUE)			
5		?C(F:=FALSE)	B1		
6			C1		

Detailed Comments:

This example describes a test that is capable of receiving an arbitrary number of C events at the lower tester PCO, until the awaited message B is received.

15.16 Constraints Reference

15.16.1 Purpose of the Constraints Reference column

This column allows references to be made to a specific constraint placed on an ASP, PDU or CM. Such constraints are defined in the constraints part (see clauses 12, 13 and 14). The constraints reference shall be present in conjunction with SEND, IMPLICIT SEND and RECEIVE. A constraints reference is optional if an ASP or CM has no parameters or if a PDU has no fields. It shall not be present with any other kind of TTCN statement.

The entry Constraints Reference column may be an actual constraint reference, the AnyValue symbol ("?"), or a formal parameter whose actual parameter shall be a constraint reference or the AnyValue symbol. If AnyValue is used in place of a constraint reference it means a "don't care" constraint, equivalent to a constraint with AnyOrNone ("*") in every parameter, field or element.

An actual constraint reference has the syntax:

SYNTAX DEFINITION:

670 ConsRef ::= ConstraintIdentifier [ActualCrefParList]
671 ActualCrefParList ::= "("ActualCrefPar { CommaActualCrefPar} ")"
672 ActualCrefPar ::= Value

EXAMPLE 99 – A constraint reference without a parameter list:

	N_SAP?CR_PDU	CRI		

15.16.2 Passing parameters in Constraint References

A constraint reference may have an optional parameter list to allow the manipulation of specific constraint values from the behaviour tree.

The actual parameter list shall fulfil the following:

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

b) each actual parameter shall evaluate to either a value of its corresponding formal type or a matching symbol that can match a value of that formal type.

If a constraint is passed as an actual parameter, and that constraint is declared with a formal parameter list, then the constraint shall also have a (possibly nested) actual parameter list. All variables appearing in the parameter list shall be bound when the constraint is used. If an unbound variable is used, then this is a test case error.

EXAMPLE 100 – A constraints reference with a parameter list:

	N_SAP?N_DATAreq	D1(P1,CR1(P2))		

Where D1 is a constraint on N_DATAreq with two parameters (actual parameters P1 and CR1), and CR1 is a constraint with one parameter (actual parameter P2).

15.16.3 Constraints and qualifiers and assignments

If an event is qualified and also has a constraints reference, this shall be interpreted as: the event matches if and only if both the qualifier *and* the constraint hold.

If an event is followed by an assignment and has a constraints reference and/or a qualifier, then this shall be interpreted as: the assignment is performed if and only if the event occurs according to the definition given above.

15.17 Verdicts

15.17.1 Introduction

Entries in the verdict column in Dynamic Behaviour tables shall be either:

– a preliminary result, which shall be given in parentheses;

– or an explicit final verdict.

An entry, of either type, shall not occur on an empty line, or on the following TTCN statements:

a) an ATTACH construct;

b) a REPEAT construct;

c) a GOTO;

d) an IMPLICIT SEND.

SYNTAX DEFINITION:

674 Verdict ::= Pass| Fail| Inconclusive| Result

675 Pass ::= **PASS**| **P**| "("**PASS**")"| "("**P**")"

676 Fail ::= **FAIL**| **F**| "("**FAIL**")"| "("**F**")"

677 Inconclusive ::= **INCONC**| **I**| "("**INCONC**")"| "("**I**")"

678 Result ::= **R**

NOTE – During Test Case execution, whenever an entry in a behaviour tree occurs for which there is a corresponding entry in the verdict column of the abstract Test Case, that verdict column information is intended to be recorded in the conformance log in such a way that it is associated with the record of that entry in the behaviour tree.

15.17.2 Preliminary results

A predefined variable called R, of the predefined type R_TYPE, is available to each Test Case to store any intermediate results. These values are predefined identifiers and as such are case sensitive.

R may be used wherever other Test Case Variables may be used, except that it shall not be used on the left-hand side of an assignment statement. Thus, it is a read-only variable, except for the changes to its value caused by entries in the verdict column (as specified below).

If a preliminary result is to be specified in the verdict column it shall be one of the following:

a) **(P)** or **(PASS)**, meaning that some aspect of the test purpose has been achieved;

b) **(I)** or **(INCONC)**, meaning that something has occurred which makes the Test Case inconclusive for some aspect of the test purpose;

c) **(F)** or **(FAIL)**, meaning that a protocol error has occurred or that some aspect of the test purpose has resulted in failure.

NOTE 1 – PASS or P, FAIL or F and INCONC or I are keywords that are used in the verdicts column only. The predefined identifiers *pass*, *fail*, *inconc* and *none* are values that represent the possible contents of the predefined variable R. These predefined identifiers are to be used for testing the variable R in behaviour lines only.

Whenever a preliminary result is recorded, because the corresponding entry in the behaviour tree is executed, then the value of the predefined Test Case Variable R shall be changed according to Table 7.

NOTE 2 – Thus, the order of precedence (lower to higher) is: N, P, I, F. Even if R has value *fail* it can be useful to record a preliminary result of P or I in order to record in the conformance log that a P or I is appropriate for some aspect of the test purpose, despite the fact that this will not change the value of R.

Table 7/X.292 – Calculation of the variable R

Current value of R	Entry in verdict column		
	(PASS)	(INCONC)	(FAIL)
None	pass	inconc	fail
Pass	pass	inconc	fail
Inconc	inconc	inconc	fail
Fail	fail	fail	fail

15.17.3 Final verdict

If an explicit final verdict is to be specified in the verdict column, it shall be one of the following:

a) **P** or **PASS**, meaning that a pass verdict is to be recorded;

b) **I** or **INCONC**, meaning that an inconclusive verdict is to be recorded;

c) **F** or **FAIL**, meaning that a fail verdict is to be recorded;

d) the predefined variable R, meaning that the value of R is to be taken as the final verdict, unless the value of R is *none* in which case a test case error is recorded instead of a final verdict.

Table 8/X.292 – Calculation of the final verdict R

Current value of R	Entry in verdict column			
	(PASS)	(INCONC)	(FAIL)	R
None	pass	inconc	fail	*error*
Pass	pass	inconc	fail	pass
Inconc	*error*	inconc	fail	inconc
Fail	*error*	*error*	fail	fail

Whenever, during execution of a Test Case, an explicit final verdict is specified, then this terminates the Test Case. For compliance with Recommendation X.291, an explicit final verdict should be specified only if the Test Case has returned to a suitable stable testing state (e.g., the idle testing state).

NOTE 1 – The termination of the Test Case caused by the specification of an explicit final verdict is necessary, for example, if the stable state is reached in an attached Test Step when subsequent behaviour is specified in the calling tree.

If the leaf of the behaviour tree is reached without an explicit final verdict being specified, then the final verdict is determined as for case d) above (i.e., as if R had been put in the verdict column).

If an explicit final verdict other than R is to be recorded, then that verdict shall be compared with the value in R to determine whether or not they are consistent. If R is *fail*, then a final verdict of **PASS** or **INCONC** shall be regarded as inconsistent; if R is *inconc*, then a final verdict of **PASS** shall be regarded as inconsistent. If there is one of these inconsistencies, then it is a test case error.

NOTE 2 – In such a case, "Test Case Error" should be recorded in the conformance log.

15.17.4 Verdicts and OTHERWISE

An OTHERWISE statement shall not lead to a PASS verdict. It should lead to a FAIL verdict, because the OTHERWISE could match an invalid test event.

15.17.5 Verdict assignment in concurrent TTCN

In concurrent TTCN, the final verdict is assigned by the MTC, either explicitly in the verdict column or implicitly as a consequence of MTC termination. Preliminary test results are maintained in the global result variable, which is accessible to the MTC as the test case variable R. The global result variable is updated whenever a preliminary result or verdict is recorded in the verdict column by a matched MTC behaviour line. If the MTC terminates without assigning an explicit verdict, then the verdict shall be determined as if R had been placed in the verdict column (see 15.17.3 d).

In addition, each PTC shall record at least one preliminary result. This preliminary result is maintained in its local result variable, which is accessible to the PTC as its test case variable R. When a preliminary result is assigned by a PTC, by any entry in the verdict column of a matched PTC behaviour line (whether or not the entry is in parentheses), both its local result variable and the global result variable are updated using the algorithm specified in 15.17.2. In a PTC, an entry in the verdict column without parentheses around it is not a final verdict, but shall cause termination of the PTC if that behaviour line matches.

Termination of the MTC before termination of all PTCs shall result in a test case error.

When the MTC uses the R variable in a Boolean expression or an assignment, it accesses the global result variable. When a PTC uses the R variable in a Boolean expression or an assignment, it accesses its local result variable. The MTC may also access a local result variable of its own by using the predefined test case variable MTC_R rather than R. MTC_R is of predefined type R_TYPE. MTC_R is updated whenever a preliminary result is recorded in the verdict column by a matched MTC behaviour line, but is unaffected by the preliminary results of PTCs. The MTC_R variable shall not be used in the verdict column.

The value of a PTC's local result variable can be communicated to another Test Component only via CMs. The value of the MTC's local or global result variables can be communicated to a PTC only via CMs.

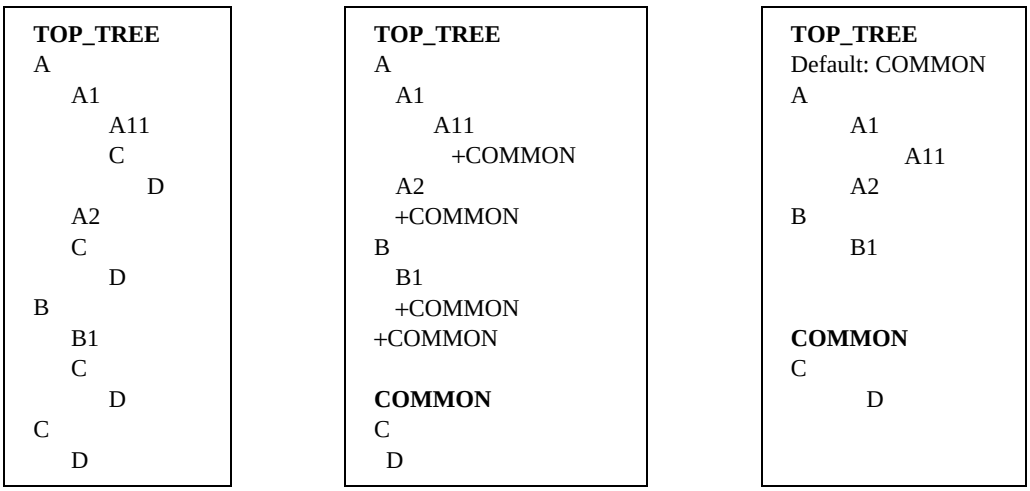
15.18 Meaning of Defaults

15.18.1 Introduction

In many cases Default behaviour will be used to emphasize a set of interesting paths through a test by declaring the less interesting common alternatives (plus their subsequent behaviour) as Default behaviour.

The same effect, though less concisely, would be achieved by Test Step attachment (e.g., +DEFAULT) as an additional general last alternative. As opposed to tree attachment, Default behaviour expands into many points of the tree it is associated with. This property calls for a careful use of Defaults.

EXAMPLE 101 – Identifying a Default tree:



1: the complete set of alternatives

2: explicit tree attachment

3: Default achieves the same as 2

No Default behaviour shall be specified to a Default behaviour, i.e., a Default may not have Default behaviour itself. Tree attachments shall not be used in Default behaviour trees, i.e., Default behaviour trees shall not attach Test Steps. Test Cases or Test Steps shall not be referred to as Defaults.

For the execution of a Test Case it is not necessary to expand Defaults everywhere in all the trees referring to them. This can be seen from an operational description of the meaning of Defaults: in attempting to match a sequence of alternatives (which may need repeated attempts), each time they all failed to match, the first level of alternatives of the Default behaviour are attempted as well. If none of these matches either, the sequence is retried with the new states of timers and queues at all PCOs concerned. If there is a match in the Default, the Default behaviour is pursued at that point.

To ensure that no subsequent behaviour will occur following the execution of a Default behaviour, the execution of a leaf of a Default tree, other than a RETURN statement, shall cause the termination of the test case. In order to accomplish this termination, in a Default tree, every leaf which has no verdict or preliminary result in the verdict column is implicitly provided with a verdict column entry of "R", and every leaf which has a preliminary result in the verdict column has that preliminary result implicitly transformed into a final verdict.

15.18.2 Default References

Test Case and Test Step behaviours reference a list of Default behaviours in the Default Library through the Default entry in the table header.

SYNTAX DEFINITION:

631 DefaultReference ::= DefaultIdentifier [ActualParList]

Each reference in this list locates a Default by its unique identifier. The DefaultIdentifier shall be a reference to a Default defined in the Default Library.

Defaults can be parameterized. The actual parameter list shall fulfil the following:

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

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

c) all variables appearing in the parameter list shall be bound when the constraint is invoked.

EXAMPLE 102 – Default reference

102.1

Test Case Dynamic Behaviour					
Test Case Name	**:** DEF_EX1				
Group	**:** TTCN_EXAMPLES/DEFAULT_EXAMPLE1/				
Purpose	**:** To illustrate the use of Defaults				
Default	**:** DEF1(L)				
Comments	**:** The tree of example ** can be split into this Test Case with the Default behaviour DEF1.				
No.	**Label**	**Behaviour Description**	**Constraint Ref**	**Verdict**	**Comments**
1		L!CONNECTrequest	CR1		Request …
2		L?CONNECTconfirm	CC1		… Confirm
3		L!DATArequest	DTR1		Send Data
4		L?DATAindication	DTI1		Receive Data
5		L!DISCONNECTrequest	DSC1	PASS	Accept

102.2

Default Dynamic Behaviour					
Default Name	**:** DEF_EX1				
Group	**:** TTCN_EXAMPLES/DEFAULTS_LIB/DEFAULT_1/				
Object	**:** Illustration of a simple Default				
Comments	**:** The tree of example ** can be split into this Test Case with the Default behaviour DEF1.				
No.	**Label**	**Behaviour Description**	**Constraint Ref**	**Verdict**	**Comments**
1		X?DISCONNECTindication	DSC2	INCONC	Premature

NOTE – Syntactically, the Default behaviour of the second of the two tables in the above example attaches X?DISCONNECTindication as an alternative to each of the L! and L? statements in the first table. However, attachment of the Default tree as an alternative to an L! statement that always succeeds is meaningless.

15.18.3 RETURN statement

The RETURN statement is an extension of the Default behaviour description capabilities. A RETURN statement shall only be used in a Default tree. It shall have the syntax:

When the Default expansion of a tree is performed, execution of a RETURN statement will cause processing to continue at the first alternative in the set of alternatives that caused the Default behaviour to be attempted.

15.18.4 ACTIVATE statement

The ACTIVATE statement allows the activation of one set of Default behaviours. Instead of being implicitly active for the duration of the test case, defaults may be activated selectively by the ACTIVATE statement. Default behaviour thus activated is attempted in the order in which it is specified by the ACTIVATE, e.g. ACTIVATE (Def_1, Def_2) will cause Def_1 to be executed before Def_2 when default behaviour is needed.

The default behaviour specified in an ACTIVATE statement overrides any active default behaviour, including default behaviour specified in a test case or test step header.

An ACTIVATE with an empty default reference list, i.e. ACTIVATE(), deactivates all default behaviour.

15.18.5 Defaults and tree attachment

Whenever tree attachment is used it is important to have a clear understanding of how Defaults apply both to the calling tree and to the attached Test Step. In order to avoid hidden side-effects, the Defaults that apply within an attached Test Step are defined to be those specified in the table that defines that Test Step. Thus, if the Test Step is defined in the Test Step Library, then the Defaults that apply are specified in header of the Test Step behaviour table. Alternatively, if the Test Step is defined locally in the same behaviour table as the calling tree, then the same Defaults apply to both the calling tree and the attached Test Step.

In order to avoid multiple insertions of Defaults within a set of alternatives, the Default specified for a particular tree do not apply to the top level of alternatives of that tree unless the tree is the root tree of a Test Case.

In order to generate a correct expansion of a tree it is necessary to expand the Defaults both:

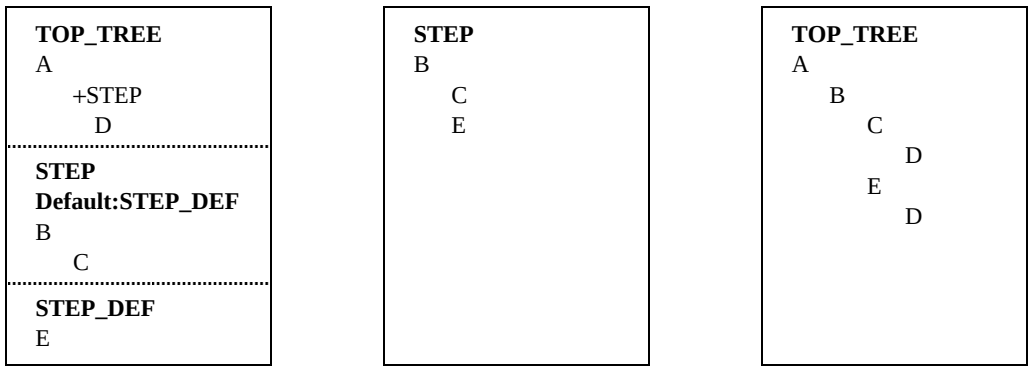
a) before the tree is expanded as an attached tree; and

b) before any of the tree's attached Test Steps are expanded.

The expansion of Defaults is thus local to a single tree and comprises the attachment of the Default tree to the bottom of every set of alternatives within the tree (except the top set of alternatives for any tree other than the root tree of a Test Case).

Default expansion rules hold equally in the case where a set of alternatives contains an OTHERWISE event.

EXAMPLE 103 – Locality of a Default against a Test Step:

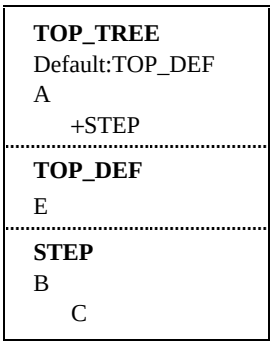


1: TOP_TREE attaches STEP, which has the Default STEP_DEF

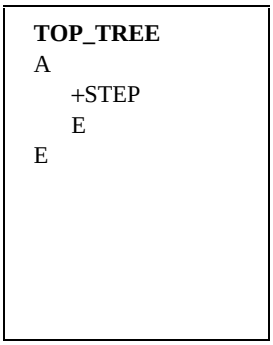
2: STEP_DEF expanded into STEP

3: STEP expanded into TOP_TREE

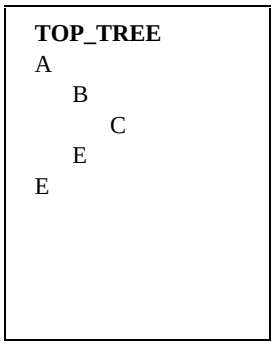
EXAMPLE 104 – Locality of a Default against a calling tree:



1: TOP_TREE attaches STEP. TOP_TREE has the Default TOP_DEF

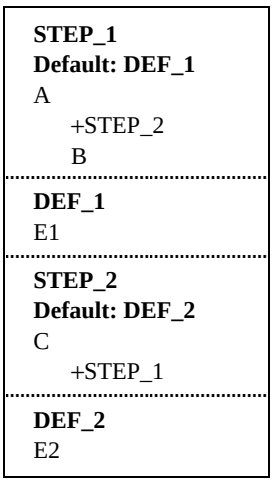


2: TOP_DEF expanded TOP_TREE

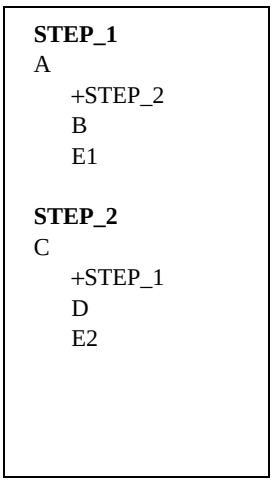


3: STEP expanded into TOP_TREE

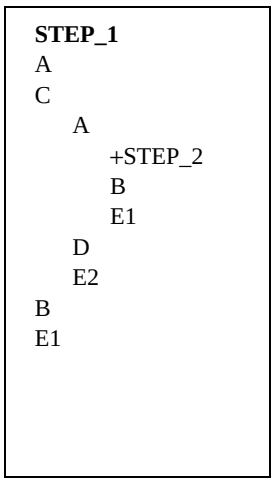
EXAMPLE 105 – A case of cyclic tree attachment:



1: STEP_1 and STEP_2 attach each other. STEP_1 has Default DEF_1. STEP_2 has Default DEF_2



2: DEF_1 expanded into STEP_1 and DEF_2 expanded into STEP_2



3: After one expansion of the Default-free STEP_2 and one expansion of the Default-free STEP_1

NOTE – Such cyclic attachments are discouraged.

15.18.6 Tree Attachment, Defaults, Activate and Return

If the ACTIVATE operation is used within a test case, the semantics of defaults and tree attachment can only be described dynamically rather than statically. Indeed, the operational semantics of defaults in Annex B are specified in terms of dynamic tree expansion, one level at a time.

In this dynamic semantic model, the specification of a list of defaults in the header is equivalent to prefixing the behaviour tree with an ACTIVATE of that list of default trees. In a test step, placing a default list in the header is equivalent to placing an ACTIVATE of that list of default trees between each alternative in the first level of alternatives and its subsequent behaviour. If a test step is attached which has no defaults specified in the header, then the implied ACTIVATE operations have no parameters and hence deactivate all defaults.

Since behaviour subsequent to a tree attachment takes its defaults from the context of the calling tree rather than attached test step, tree attachment implies the insertion of an ACTIVATE after every non-terminating leaf node (i.e. one which does not assign a verdict) to restore the defaults to those of the context in which the attachment was made. In the case of the leaf node being a RETURN, this implies ACTIVATE has to come before the RETURN to ensure that it takes effect before jumping back into the outer context.

The effect of a combination of defaults and tree attachment is illustrated by the example test case shown in Example 106.

EXAMPLE 106 – Example test case X-Def1 to illustrate the meaning of defaults:

Test Case Dynamic Behaviour			
Test Step Name : X-Def1 **Group :** **Purpose :** **Default :** D1, D2			
L	**Behaviour Description**	**Cref**	**V**
	X +T1 Y Z +T2		

Test Case Dynamic Behaviour			
Test Step Name : T1 **Group :** **Objective :** **Default :** D3, D4			
L	**Behaviour Description**	**Cref**	**V**
	A B C		

Test Case Dynamic Behaviour			
Test Step Name : T2 **Group :** **Objective :** **Default :**			
L	**Behaviour Description**	**Cref**	**V**
	D E F		

This example test case is equivalent to the one shown in Example 107, in which the list of defaults in the test case header has been replaced by an ACTIVATE of the same list of defaults as the first TTCN statement of the behaviour tree.

EXAMPLE 107 – Alternative specification of example test case X-Def1 using ACTIVATE:

Test Case Dynamic Behaviour			
Test Step Name : X-Def1 **Group :** **Purpose :** **Default :**			
L	**Behaviour Description**	**Cref**	**V**
	ACTIVATE(D1,D2) X +T1 Y Z +T2		

The processing of an ACTIVATE sets the current default context. Progression to the next level of alternatives attaches the list of default trees in the current default context to the next level of alternatives.

Thus, the evaluation of the example test case shown in Example 107 could progress as illustrated in Figure 8. Firstly, the ACTIVATE(D1,D2) statement is evaluated to set the default context to D1 and D2. Then, assuming that X matches, D1 and D2 are attached at the same level of alternatives as T1. When T1 is then expanded, ACTIVATE(D3,D4) is inserted after the first level of alternatives of that test step, and ACTIVATE(D1,D2) is inserted after the two leaf nodes in order to restore the default context before the subsequent behaviour, Y, is reached. Assuming that A then matches, the defaults D1 and D2 are attached redundantly at the same level of alternatives as the ACTIVATE; this is because the current default context is always appended to the next level of alternatives, indiscriminately, even if the next level of alternatives consists of a construct or pseudo-event which always matches. When the new ACTIVATE statement is evaluated, the default context is changed to that applicable to test step T1. Then if B matches, the evaluation progresses to the ACTIVATE which restores the default context back to that applicable to the root tree.

Example 108 gives another example test case, this one mixing defaults specified in headers with an explicit ACTIVATE statement and tree attachment.

EXAMPLE 108 – Example test case X-Def2 to illustrate the meaning of defaults and ACTIVATE:

Test Case Dynamic Behaviour			
Test Step Name : X-Def2 **Group :** **Purpose :** **Default :** D1			
L	**Behaviour Description**	**Cref**	**V**
	X ACTIVATE(D2) +T S +T S		

Test Case Dynamic Behaviour			
Test Step Name : T **Group :** **Objective :** **Default :** D3			
L	**Behaviour Description**	**Cref**	**V**
	Y Z		

The progression of the evaluation of this test case is illustrated in Figure 9. This shows the progression of the evaluation through the two main paths of the test case, showing that the default context applicable to the first S is determined by the ACTIVATE, whereas the default context applicable to the second S is determined by the defaults specified in the test case header; neither of these default contexts for the S statements is affected by the preceding tree attachments.

Figure 9 begins by showing the effect of expanding the attachment of T at the first level of alternatives plus the appending of the initial defaults. If X matches, the evaluation progresses via the ACTIVATE(D2) to the second occurrence of the attachment of T, with the default context changed to D2 and the attachment of D2 appended at the same level of alternatives as T. T is then expanded, remembering to insert the two ACTIVATE statement to set the test step default context and then restore the root tree default context. These changes in the default context are then shown in the next two stages of the evaluation, assuming that first Y matches and then Z. The result is S with an alternative of the attachment of D2 being evaluated in default context D2.

The alternative path shown in Figure 9 starts with Y matching instead of X. This causes the progression into default context D3, whereupon if Z matches the default context is restored to be D1. Thus, what is reached down this path of the progression is S with an alternative of the attachment of D1 being evaluated in default context D1.

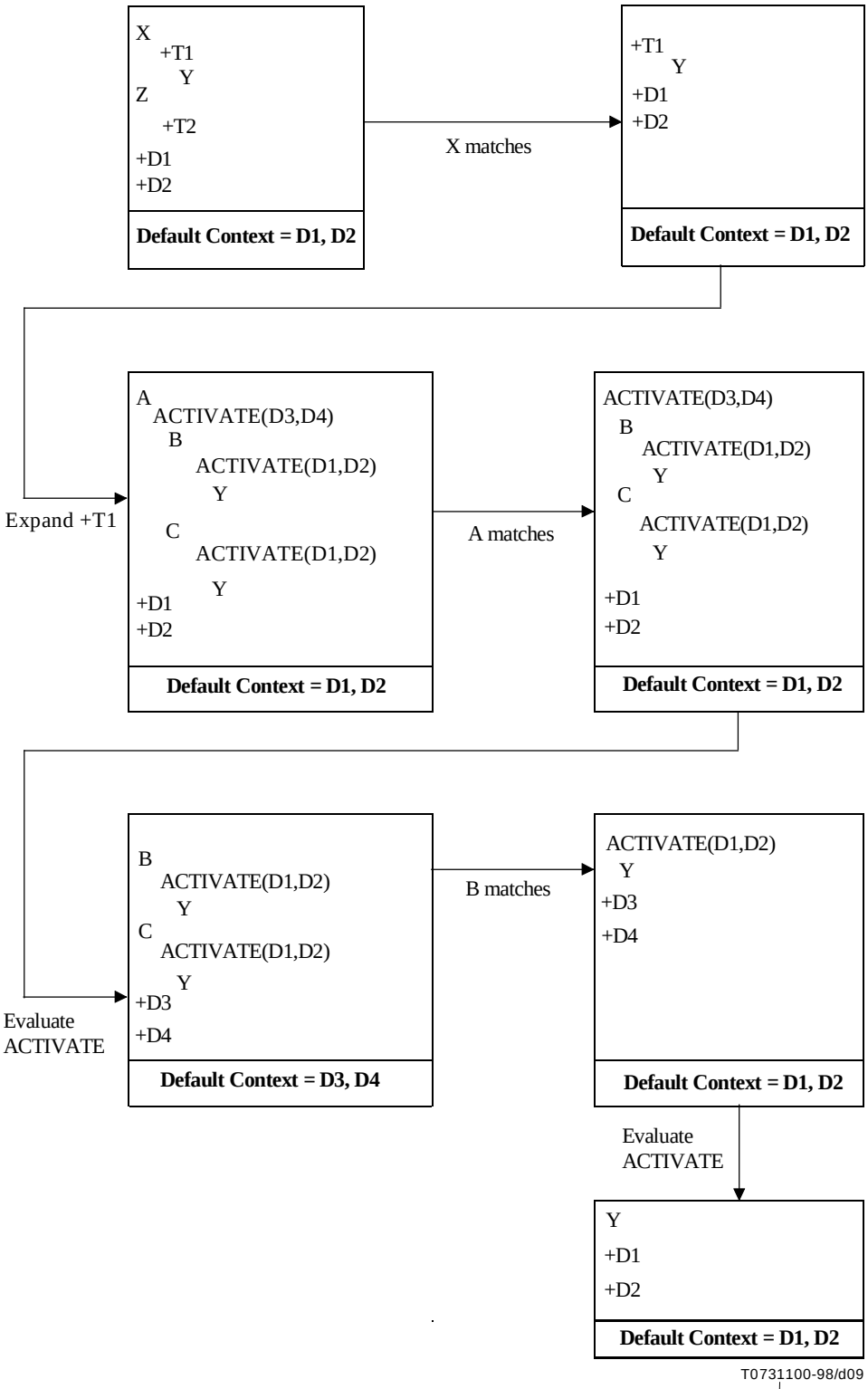


Figure 8/X.292 – Possible progression of evaluation of example test case X-Def1

The progression of evaluation of example test cases in Figures 8 and 9 has not shown the expansion of the default trees. If when the default tree is expanded, it is found that the default tree or any associated local tree contains a RETURN construct, this is equivalent to a label being placed at the head of the current set of alternatives with every RETURN construct being replaced by an ACTIVATE, to restore the default context of the calling tree, followed by a GOTO construct to go to that new label.

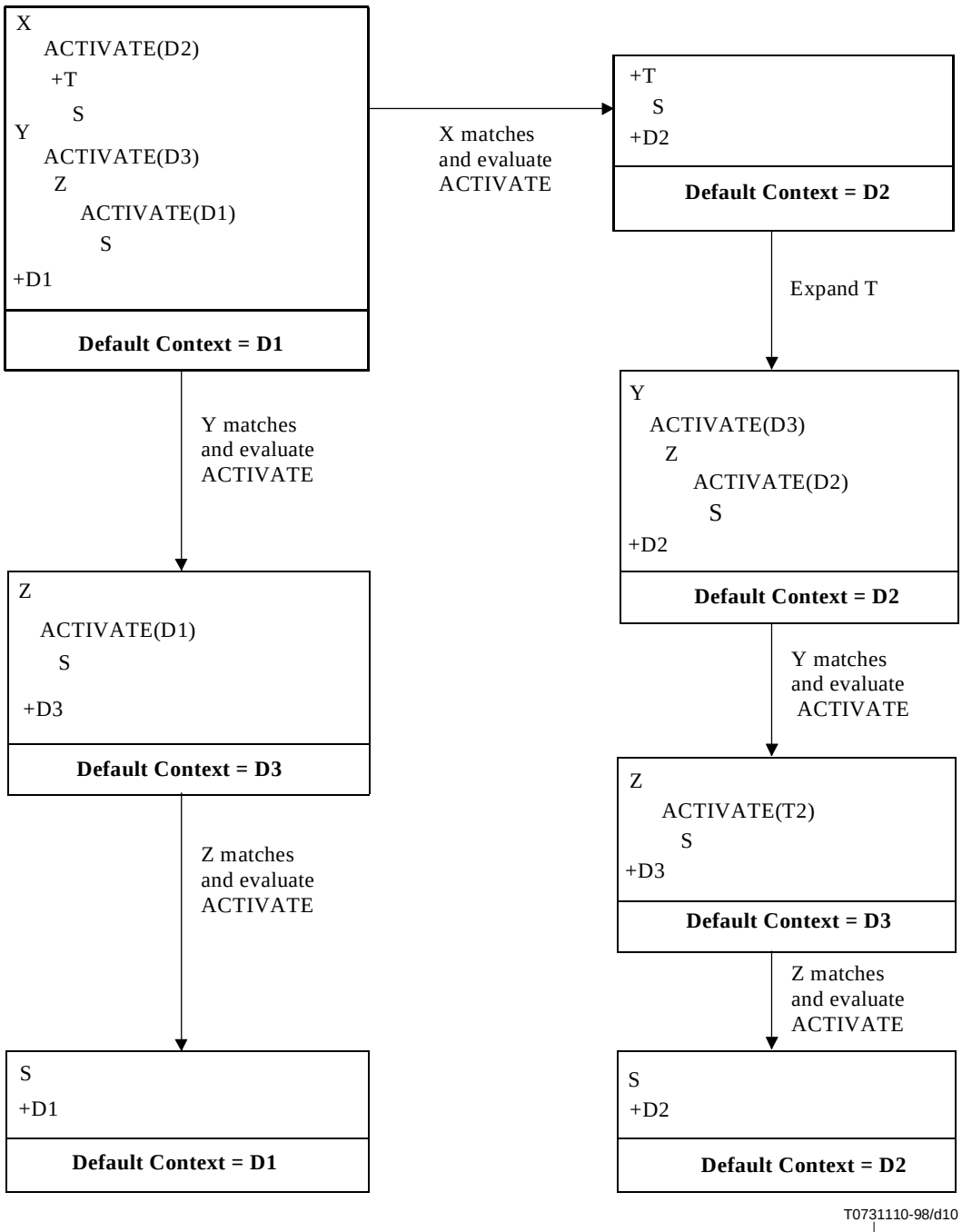


Figure 9/X.292 – Possible progression of evaluation of example test case X-Def2

All leaf nodes, other than RETURN, of a default behaviour tree in which all local subtrees have been attached have no subsequent behaviour and so they shall either set a verdict or result in a test case error.

To illustrate this, the example test case given in Example 109 will be used.

EXAMPLE 109 – Example test case X-Def3 to illustrate the meaning of defaults and RETURN:

Test Case Dynamic Behaviour			
Test Step Name : X-Def3 **Group :** **Purpose :** **Default :** D1			
L	**Behaviour Description**	**Cref**	**V**
	X Y		P

Test Case Dynamic Behaviour			
Default Name : D1 **Test Step Name :** **Objective :**			
L	**Behaviour Description**	**Cref**	**V**
	C D RETURN E		F

The progression of the evaluation of this example test case is illustrated in Figure 10. Firstly, the default tree D1 is attached at the first level of alternatives of the root tree. D1 is then expanded. Since D1 contains a RETURN statement, this is a fairly complex expansion. The top event in the level of alternatives at which the attachment occurs is labelled with a unique label, L. Since the attached tree is a default, its own internal default context is empty because defaults do not have their own defaults, and therefore an ACTIVATE with no arguments is inserted after the first level of alternatives of the attached tree. In addition the RETURN statement is replaced by an ACTIVATE to restore the default context to D1, followed at the next level by GOTO L. Now, when this expanded tree is evaluated, if C matches, it progresses to the ACTIVATE() statement together with the redundant attachment of the default context, D1. The effect of evaluating the ACTIVATE() is to empty the default context. Then, if D matches, the ACTIVATE(D1) is evaluated to restore the default context to D1. This leads to the GOTO statement together with another redundant attachment of the default context D1. The evaluation of the GOTO then returns the processing to the state in which the label L was added. Evaluation will continue to cycle round this loop until either X, followed by Y, matches for a pass, or C, followed by E, matches for a fail.

15.18.7 Defaults and CREATE

Default behaviour is not inherited by test steps which are used in a CREATE operation, i.e. test steps which execute their behaviour description in parallel with the MTC. Thus, the scope of Default behaviour in concurrent TTCN is always local to the MTC or a PTC.

In instances when a test step is used in a CREATE operation, the Default behaviour specified in the test step header shall be applied at the first level of indentation. This use of Defaults is consistent with the application of Defaults in test cases.

15.18.8 Defaults and CMs

Default behaviour is applied to a set of alternatives which receive only CMs. This may cause PDUs which arrive prior to receipt of the executed CM, or PDUs which are already in the PCO queue but not yet received, to be removed from the PCO queue. To prevent the removal of PDUs from the PCO queue, the NO_DEFAULTS construct shall be specified as the event immediately preceding the set of alternatives which receive only the CM(s).

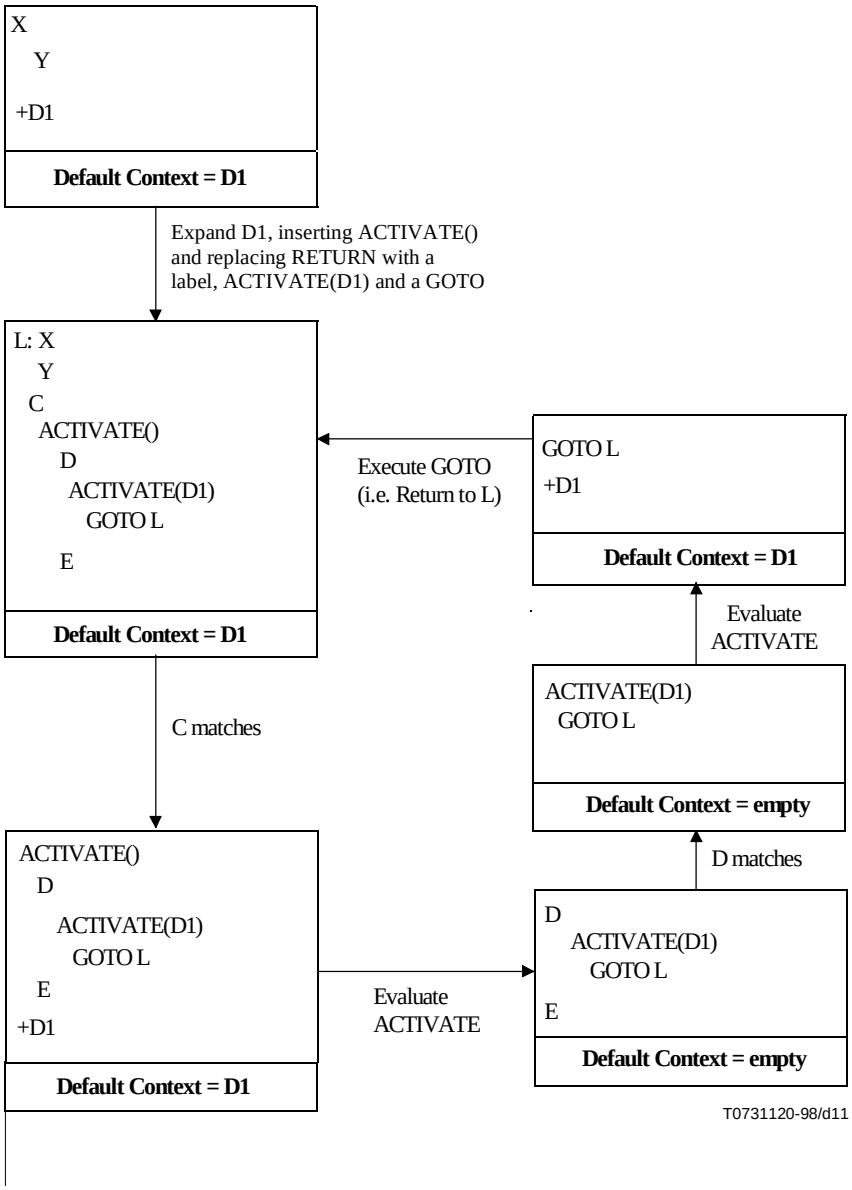


Figure 10/X.292 – Possible progression of evaluation of example test case X-Def3

16 Page continuation

16.1 Page continuation of TTCN tables

When any TTCN table is too long to fit on a single page the following mechanism shall be used:

a) the words "Continued on next page" shall be printed *after* the table line where the split occurs;

b) the words "Continued from previous page" shall be printed *before* the continued table on the next page.

Tables may be split at any location, i.e., in their header, body, or footer section. In all cases, the sections title (e.g., column headers), shall be repeated on the next page. The complete header may or may not be repeated.

EXAMPLE 110 – A continued Test Suite Parameters table:

Test Suite Parameter Declarations			
Parameter Name	**Type**	**PIC/PIXIT Ref**	**Comments**
PAR1	INTEGER	PICS question aa	
PAR2	BOOLEAN	PICS question bb	
PAR3	IA5String	PICS question cc	

Continued on next page page n

Continued from previous page page n + 1

Test Suite Parameter Declarations			
Parameter Name	**Type**	**PIC/PIXIT Ref**	**Comments**
PAR4	BOOLEAN	PICS question dd	
PAR5	HEXSTRING	PICS question ee	

16.2 Page continuation of dynamic behaviour tables

When it is necessary to continue a dynamic behaviour table, then either of the following two mechanisms can be used:

a) modularization,

where some part of the behaviour of the tree is specified as a library (non-local) Test Step, thereby modularizing the tree and reducing the amount of behaviour for the current proforma to that which will fit on a single page; or

b) page continuation mechanism,

where, in the case of a dynamic behaviour table, in order to aid alignment of indentation levels, the following additional information shall be presented:

1) the level of indentation (enclosed in square brackets) of the last TTCN statement before the page split occurs, shall be printed before the words "Continued on next page";

2) on the continued page, the level of indentation (enclosed in square brackets) of the first TTCN statement in the continued table, shall be printed after the words "Continued from previous page".

It may be necessary in the case of lengthy Test Cases to indent to a different level than the stated one. In such cases the stated level of indentation enclosed in square brackets will be aligned with the chosen indentation of the first statement line in the continued table. To further aid alignment of indentation levels, additional indications of indentation levels may also be given.

Annex A

Syntax and static semantics of TTCN

A.1 Introduction

This annex defines the syntax and the static semantics of TTCN. There are two forms of TTCN, a graphical form (TTCN.GR) and a machine processable form (TTCN.MP). For the human user the graphical form of TTCN, the TTCN.GR, takes advantage of an easily understood visual interpretation. However, TTCN.GR does not readily lend itself to machine processing. The TTCN.MP addresses this problem and serves the following purposes:

a) to provide a formal syntax for TTCN in BNF;

b) to act as a transfer syntax;

c) to ease automated derivation of ETSs from ATSs;

d) other machine processing.

NOTE – Automated derivation of ETSs is outside the scope of this Recommendation.

This annex also defines the static semantics for both TTCN.GR and TTCN.MP.

A.2 Conventions for the syntax description

A.2.1 Syntactic metanotation

Table A.1 defines the metanotation used to specify the extended form of BNF grammar for TTCN (henceforth called BNF):

In the metanotation, concatenation binds more tightly than the alternative operator. Hence "abc def | ghi jkl" is equivalent to "(abc def) | (ghi jkl)".

Table A.1/X.292 – TTCN.MP Syntactic Metanotation

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

A.2.2 TTCN.MP syntax definitions

A.2.2.1 Complete tables defined in TTCN.GR are represented in TTCN.MP by productions of the kind:

$Begin_KEYWORD .... $End_KEYWORD

EXAMPLE A.1 – TS_PARdcls ::= **$Begin_TS_PARdcls** {TS_PARdcl}+ **$End_TS_PARdcls**

Normally, these productions contain at least one mandatory component.

A.2.2.2 Both sets of lines of a table and individual lines (i.e., sets of fields in a table) are represented by productions of the kind:

$KEYWORD .... $End_KEYWORD

Begin does not appear in the opening keyword.

EXAMPLE A.2 – TS_PARdcl ::= **$TS_PARdcl** TS_PARid TS_PARtype PICS_PIXIT [Comment]
$End_TS_PARdcl

A.2.2.3 Individual fields in a line are represented by:

$KEYWORD ....

There is no closing keyword.

EXAMPLE A.3 – TS_ParId ::= **$TS_ParId** TS_ParIdentifier

EXAMPLE A.4 – TS_ParIdentifier ::= Identifier

A.2.2.4 Sets of tables, up to and including the test suite, are represented by productions of the kind:

$KEYWORD .... $End_KEYWORD

EXAMPLE A.5 – ASP_TypeDefs ::= **$ASP_TypeDefs** [TTCN_ASP_TypeDefs] [ASN1_ASP_TypeDefs]
$End_ASP_TypeDefs

A.2.2.5 All other productions defining non-terminal symbols have no keywords at the beginning or the end of the right-hand expression.

EXAMPLE A.6 – TimerIdentifier ::= Identifier

A.2.2.6 When parsing TTCN.MP, any symbol not allowed within an identifier may denote the end of an identifier. In those cases in which it is necessary to insert a meaningless character at the end of an identifier in order to separate it from another identifier or keyword (e.g. when an identifier is followed by a keyword such as **BY** or **OR**), then the recommended separators are space and tab characters.

A.3 TTCN.MP syntax productions in BNF

A.3.1 TTCN Specification

1 TTCN_Specification ::= TTCN_Module | Suite

A.3.2 TTCN Module

2 TTCN_Module ::= **$TTCN_Module** TTCN_ModuleId TTCN_ModuleOverviewPart [TTCN_ModuleImportPart] [DeclarationsPart] [ConstraintsPart] [DynamicPart] **$End_TTCN_Module**

3 **TTCN_ModuleId ::=$TTCN_ModuleId** TTCN_ModuleIdentifier

4 TTCN_ModuleIdentifier ::= Identifier

A.3.2.1 TTCN Module Overview Part

5 TTCN_ModuleOverviewPart ::= **$TTCN_ModuleOverviewPart** TTCN_ModuleExports [TTCN_ModuleStructure] [TestCaseIndex] [TestStepIndex] [DefaultIndex] **$End_TTCN_ModuleOverviewPart**

A.3.2.1.1 TTCN Module Exports

6 TTCN_ModuleExports ::= **$Begin_TTCN_ModuleExports** TTCN_ModuleId [TTCN_ModuleRef] [TTCN_ModuleObjective] [StandardsRef] [PICSref] [PIXITref] [TestMethods] [Comment] ExportedObjects [Comment] **$End_TTCN_ModuleExports**

7 **TTCN_ModuleRef ::= $TTCN_ModuleRef** BoundedFreeText

8 TTCN_ModuleObjective ::= **$TTCN_ModuleObjective** BoundedFreeText

9 ExportedObjects ::= **$ExportedObjects** {ExportedObject} **$End_ExportedObjects**

10 **ExportedObject ::= $ExportedObject** ObjectId ObjectType [SourceInfo] [Comment] **$End_ExportedObject**

11 **ObjectId ::= $ObjectId** ObjectIdentifier

12 ObjectIdentifier ::= Identifier | ObjectTypeReference

13 ObjectTypeReference ::= Identifier "[" Identifier "]"

/* STATIC SEMANTICS – The first Identifier is a NamedNumber or an Enumeration and the Identifier contained in brackets is the name of the corresponding type. */

14 ObjectType ::= **$ObjectType** TTCN_ObjectType

15 TTCN_ObjectType ::= **SimpleType_Object** | **StructType_Object** | **ASN1_Type_Object** | **TS_Op_Object** | **TS_Proc_Object** | **TS_Par_Object** | **SelectExpr_Object** | **TS_Const_Object** | **TS_Var_Object** | **TC_Var_Object** | **PCO_Type_Object** | **PCO_Object** | **CP_Object** | **Timer_Object** | **TComp_Object** | **TCompConfig_Object** | **TTCN_ASP_Type_Object** | **ASN1_ASP_Type_Object** | **TTCN_PDU_Type_Object** | **ASN1_PDU_Type_Object** | **TTCN_CM_Type_Object** | **ASN1_CM_Type_Object** | **EncodingRule_Object** | **EncodingVariation_Object** | **InvalidFieldEncoding_Object** | **Alias_Object** | **StructTypeConstraint_Object** | **ASN1_TypeConstraint_Object** | **TTCN_ASP_Constraint_Object** | **ASN1_ASP_Constraint_Object** | **TTCN_PDU_constraint_Object** | **ASN1_PDU_Constraint_Object** | **TTCN_CM_Constraint_Object** | **ASN1_CM_Constraint_Object** | **TestCase_Object** | **TestStep_Object** | **Default_Object** | **NamedNumber_Object** | **Enumeration_Object**

16 **SourceInfo ::= $SourceInfo** (SourceIdentifier | ObjectDirective)
/* STATIC SEMANTICS – The SourceIdentifier is the name of the original source object . */

17 SourceIdentifier ::= SuiteIdentifier | TTCN_ModuleIdentifier

18 ObjectDirective ::= Omit | **EXTERNAL**

A.3.2.1.2 TTCN Module Structure

19 TTCN_ModuleStructure ::= **$Begin_TTCN_ModuleStructure** Structure&Objectives [Comment] **$End_TTCN_ModuleStructure**

A.3.2.2 TTCN Module Import Part

20 TTCN_ModuleImportPart ::= **$TTCN_ModuleImportPart** [ExternalObjects] [ImportDeclarations] **$End_TTCN_ModuleImportPart**

A.3.2.2.1 External Objects

21 ExternalObjects ::= **$Begin_ExternalObjects** [ExternalGroupId] {ExternalObject}+ [Comment] **$End_ExternalObjects**

22 **ExternalGroupId ::= $ExternalGroupId** ExternalGroupIdentifier

23 ExternalGroupIdentifier ::= Identifier

24 ExternalObject ::= **$ExternalObject** ExternalObjectId ObjectType [Comment] **$End_ExternalObject**

25 **ExternalObjectId ::= $ExternalObjectId** ExternalObjectIdentifier

26 ExternalObjectIdentifier ::= ObjectIdentifier | TS_OpId&ParList | ConsId&ParList | TestStepId&ParList

A.3.2.2.2 Import Declarations

27 ImportDeclarations ::= **$ImportDeclarations** {ImportsOrGroup}+ **$End_ImportDeclarations**

28 **ImportsOrGroup ::= Imports | ImportsGroup**

29 ImportsGroup ::= **$ImportsGroup** ImportsGroupId {ImportsOrGroup}+ **$End_ImportsGroup**

30 ImportsGroupId ::= **$ImportsGroupId** ImportsGroupIdentifier

31 Imports ::= **$Begin_Imports** SourceId [ImportsGroupRef] [SourceRef] [StandardsRef] [Comment] ImportedObjects [Comment] **$End_Imports**

32 **SourceId ::= $SourceId** SourceIdentifier

33 ImportsGroupRef ::= **$ImportsGroupRef** ImportsGroupReference

34 ImportsGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ImportsGroupIdentifier "/"}

35 ImportsGroupIdentifier ::= Identifier

36 SourceRef ::= **$SourceRef** BoundedFreeText

37 ImportedObjects ::= **$ImportedObjects** {ImportedObject}+ **$End_ImportedObjects**

38 **ImportedObject ::= $ImportedObject** ObjectId ObjectType [SourceInfo] [Comment] **$End_ImportedObject**

A.3.3 Test suite

39 Suite ::= **$Suite** SuiteId SuiteOverviewPart [ImportPart] DeclarationsPart ConstraintsPart DynamicPart **$End_Suite**
/* STATIC SEMANTICS – SuiteId shall be the same as the SuiteId declared in TestSuiteStructure table (Suite Structure). */

40 SuiteId ::= **$SuiteId** SuiteIdentifier

41 SuiteIdentifier ::= Identifier

A.3.3.1 Test Suite Overview

42 SuiteOverviewPart::= **$SuiteOverviewPart** [TestSuiteIndex] SuiteStructure TestCaseIndex [TestStepIndex] [DefaultIndex] [TestSuiteExports] **$End_SuiteOverviewPart**

A.3.3.2 Test Suite Index

43 TestSuiteIndex ::= **$Begin_TestSuiteIndex** {ObjectInfo} [Comment] **$End_TestSuiteIndex**

A.3.3.2.1 Imported Object Info

44 ObjectInfo ::= **$ObjectInfo** ObjectId ObjectType SourceId OrigObjectId [PageNum] [Comment] **$End_ObjectInfo**

45 PageNum ::= $PageNum PageNumber

46 PageNumber ::= Number

47 OrigObjectId ::= **$OrigObjectId** ObjectIdentifier

A.3.3.3 Test Suite Structure

48 SuiteStructure ::= **$Begin_SuiteStructure** SuiteId StandardsRef PICSref PIXITref TestMethods [Comment] Structure&Objectives [Comment] **$End_SuiteStructure**

49 StandardsRef ::= **$StandardsRef** BoundedFreeText

50 PICSref ::= **$PICSref** BoundedFreeText

51 PIXITref ::= **$PIXITref** BoundedFreeText

52 TestMethods ::= **$TestMethods** BoundedFreeText

53 Comment ::= **$Comment** [BoundedFreeText]

54 Structure&Objectives ::= **$Structure&Objectives** {Structure&Objective} **$End_Structure&Objectives**

55 Structure&Objective ::= **$Structure&Objective** TestGroupRef SelExprId Objective **$End_Structure&Objective**

56 SelExprId ::=**$SelectExprId** [SelectExprIdentifier]

A.3.3.4 Test Case Index

57 TestCaseIndex ::= **$Begin_TestCaseIndex** {[CollComment] CaseIndex}+ [Comment] **$End_TestCaseIndex**

/* NOTE – Collective comments may be used in this table according to Figure 2. */

58 CollComment ::= **$CollComment** [BoundedFreeText]

59 CaseIndex ::= **$CaseIndex** TestGroupRef TestCaseId SelExprId Description **$End_CaseIndex**

/* STATIC SEMANTICS – Test Cases shall be listed in the order that they exist in the dynamic part. */

/* STATIC SEMANTICS – An explicit TestGroupReference shall be provided for the first TestCase of each TestGroup. */

/* STATIC SEMANTICS – An explicit TestGroupReference shall be provided for each TestCase that immediately follows a TestGroup. */

60 Description ::= **$Description** BoundedFreeText

A.3.3.5 Test Step Index

61 TestStepIndex ::= **$Begin_TestStepIndex** {[CollComment] StepIndex} [Comment] **$End_TestStepIndex**

/* NOTE – Collective comments may be used in this table according to Figure 2. */

62 StepIndex ::= **$StepIndex** TestStepRef TestStepId Description **$End_StepIndex**

/* STATIC SEMANTICS – TestStepId shall not include a formal parameter list. */

/* STATIC SEMANTICS – Test Steps shall be listed in the order that they exist in the dynamic part. */

/* STATIC SEMANTICS – An explicit TestStepGroupReference shall be provided for the first TestStep of each TestStepGroup. */

/* STATIC SEMANTICS An explicit TestStepGroupReference shall be provided for each TestStep that immediately follows a TestStepGroup. */

A.3.3.6 Default Index

63 DefaultIndex ::= **$Begin_DefaultIndex** {[CollComment] DefIndex} [Comment] **$End_DefaultIndex**

/* NOTE – Collective comments may be used in this table according to Figure 2. */

64 DefIndex ::= **$DefIndex** DefaultRef DefaultId Description **$End_DefIndex**

/* STATIC SEMANTICS – DefaultId shall not include a formal parameter list. */

/* STATIC SEMANTICS – Defaults shall be listed in the order that they exist in the dynamic part. */

/* STATIC SEMANTICS – An explicit DefaultGroupReference shall be provided for the first Default of each DefaultGroup. */

/* STATIC SEMANTICS – An explicit DefaultGroupReference shall be provided for eachDefault that immediately follows a DefaultGroup. */

A.3.3.7 Test Suite Exports

65 TestSuiteExports::= **$Begin_TestSuiteExports** ExportedObjects [Comment] **$End_TestSuiteExports**

A.3.3.8 Import Part

66 ImportPart ::= **$ImportPart** ImportDeclarations **$End_ImportPart**

A.3.3.9 Declarations Part

67 DeclarationsPart ::= **$DeclarationsPart** Definitions Parameterization&Selection Declarations ComplexDefinitions **$End_DeclarationsPart**

A.3.3.10 Definitions

A.3.3.10.1 General

68 Definitions ::= [TS_TypeDefs] [EncodingDefs] [TS_OpDefs] [TS_ProcDefs]

A.3.3.10.2 Test Suite Type Definitions

69 TS_TypeDefs ::= **$TS_TypeDefs** [SimpleTypeDefsOrGroup] [StructTypeDefs] [ASN1_TypeDefs] [ASN1_TypeRefsOrGroup] **$End_TS_TypeDefs**

A.3.3.10.3 Simple Type Definitions

70 SimpleTypeDefsOrGroup ::= SimpleTypeDefs | SimpleTypeGroup
71 SimpleTypeGroup ::= **$SimpleTypeGroup** SimpleTypeGroupId {SimpleTypeDefsOrGroup}+ **$End_SimpleTypeGroup**
72 SimpleTypeGroupId ::= **$SimpleTypeGroupId** SimpleTypeGroupIdentifier
73 SimpleTypeDefs ::= **$Begin_SimpleTypeDefs** [SimpleTypeGroupRef] {[CollComment] SimpleTypeDef}+ [Comment] **$End_SimpleTypeDefs**
/* NOTE – Collective comments may be used in this table according to Figure 2. */
74 SimpleTypeGroupRef ::= **$SimpleTypeGroupRef** SimpleTypeGroupReference
75 SimpleTypeGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {SimpleTypeGroupIdentifier "/"}
76 SimpleTypeGroupIdentifier ::= Identifier
77 SimpleTypeDef ::= **$SimpleTypeDef** SimpleTypeId SimpleTypeDefinition [PDU_FieldEncoding] [Comment] **$End_SimpleTypeDef**
78 SimpleTypeId ::= **$SimpleTypeId** SimpleTypeIdentifier
79 SimpleTypeIdentifier ::= Identifier
80 SimpleTypeDefinition ::= **$SimpleTypeDefinition** Type&Restriction
/* STATIC SEMANTICS – There shall be no recursive references (neither directly nor indirectly) in Type&Restriction. */
81 Type&Restriction ::= Type [Restriction]
/* STATIC SEMANTICS – Type shall be either PredefinedType or SimpleType. */
82 Restriction ::= LengthRestriction | IntegerRange | SimpleValueList
/* STATIC SEMANTICS – The set of values defined by Restriction shall be a true subset of the values of the base type. */
83 LengthRestriction ::= SingleTypeLength | RangeTypeLength
/* STATIC SEMANTICS – LengthRestriction shall be provided only when the base type is a string type (i.e. BITSTRING, HEXSTRING, OCTETSTRING or CharacterString) or derived from a string type. */
84 SingleTypeLength ::= "[" Number "]"
85 RangeTypeLength ::= "[" LowerTypeBound To UpperTypeBound "]"
/* STATIC SEMANTICS – LowerTypeBound shall be a non-negative number. */
/* STATIC SEMANTICS – LowerTypeBound shall be less than UpperTypeBound. */
86 IntegerRange ::= "(" LowerTypeBound To UpperTypeBound ")"
/* STATIC SEMANTICS – LowerTypeBound shall be less than UpperTypeBound. */
87 LowerTypeBound ::= [Minus] Number | Minus **INFINITY**
88 UpperTypeBound ::= [Minus] Number | **INFINITY**
89 To ::= **TO** | ".."
90 SimpleValueList ::= "(" [Minus] LiteralValue {Comma [Minus] LiteralValue} ")"
/* STATIC SEMANTICS – If Minus is used in SimpleValueList, then LiteralValue shall be a number. */
/* STATIC SEMANTICS – The LiteralValues shall be of the base type and shall be a true subset of the values defined by the base type. */

A.3.3.10.4 Structured Type Definitions

91 StructTypeDefs ::= **$StructTypeDefs** {StructTypeDefOrGroup}+ **$End_StructTypeDefs**
92 StructTypeDefOrGroup ::= StructTypeDef | StructTypeGroup
93 StructTypeGroup ::= **$StructTypeGroup** StructTypeGroupId {StructTypeDefOrGroup}+ **$End_StructTypeGroup**
94 StructTypeGroupId ::= **$StructTypeGroupId** StructTypeGroupIdentifier
95 StructTypeDef ::= **$Begin_StructTypeDef** StructId [StructTypeGroupRef] [EncVariationId] [Comment] ElemDcls [Comment] **$End_StructTypeDef**
96 StructId ::= **$StructId** StructId&FullId
97 StructId&FullId ::= StructIdentifier [FullIdentifier]

98 FullIdentifier ::= "(" BoundedFreeText ")"

/* STATIC SEMANTICS – Some TTCN objects allow names, as given in the appropriate protocol standard to be abbreviated. If an abbreviation is used, then FullIdentifier shall be given in the declaration of the object. */

99 StructIdentifier ::= Identifier

100 StructTypeGroupRef ::= **$StructTypeGroupRef** StructTypeGroupReference

101 StructTypeGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {StructTypeGroupIdentifier "/"}

102 StructTypeGroupIdentifier ::= Identifier

103 ElemDcls ::= **$ElemDcls** {ElemDcl}+ **$End_ElemDcls**

104 ElemDcl ::= **$ElemDcl** ElemId ElemType [PDU_FieldEncoding] [Comment] **$End_ElemDcl**

105 ElemId ::= **$ElemId** ElemId&FullId

106 ElemId&FullId ::= ElemIdentifier [FullIdentifier]

107 ElemIdentifier ::= Identifier

108 ElemType ::= **$ElemType** Type&Attributes

/* STATIC SEMANTICS – There shall be no recursive references (neither directly nor indirectly) in Type&Attributes. */

/* STATIC SEMANTICS – A structure element Type shall be a PredefinedType, TS_TypeIdentifier, PDU_Identifier, or PDU. */

A.3.3.10.5 ASN.1 Type Definitions

109 ASN1_TypeDefs ::= **$ASN1_TypeDefs** {ASN1_TypeDefOrGroup}+ **$End_ASN1_TypeDefs**

110 ASN1_TypeDefOrGroup ::= ASN1_TypeDef | ASN1_TypeGroup

111 ASN1_TypeGroup ::= **$ASN1_TypeGroup** ASN1_TypeGroupId {ASN1_TypeDefOrGroup}+ **$End_ASN1_TypeGroup**

112 ASN1_TypeGroupId ::= **$ASN1_TypeGroupId** ASN1_TypeGroupIdentifier

113 ASN1_TypeDef ::= **$Begin_ASN1_TypeDef** ASN1_TypeId [ASN1_TypeGroupRef] [EncVariationId] [Comment] ASN1_TypeDefinition [Comment] **$End_ASN1_TypeDef**

114 ASN1_TypeId ::= **$ASN1_TypeId** ASN1_TypeId&FullId

115 ASN1_TypeId&FullId ::= ASN1_TypeIdentifier [FullIdentifier]

116 ASN1_TypeIdentifier ::= Identifier

117 ASN1_TypeGroupRef ::= **$ASN1_TypeGroupRef** ASN1_TypeGroupReference

118 ASN1_TypeGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ASN1_TypeGroupIdentifier "/"}

119 ASN1_TypeGroupIdentifier ::= Identifier

120 ASN1_TypeDefinition ::= **$ASN1_TypeDefinition** ASN1_Type&LocalTypes **$End_ASN1_TypeDefinition**

121 ASN1_Type&LocalTypes ::= ASN1_Type {ASN1_LocalType}

/* STATIC SEMANTICS – Types referred to from the ASN1_Type definition shall be defined in other ASN.1 type definition tables, be defined by reference in the ASN.1 type reference table or be defined locally (i.e., ASN1_LocalTypes) in the same table, following the first type definition. */

/* STATIC SEMANTICS – ASN1_LocalTypes shall not be used in other parts of the test suite. */

122 ASN1_Type ::= Type

/* REFERENCE – Where Type is a non-terminal defined in Recommendations X.680:

Type ::= BuiltinType | ReferencedType | ConstrainedType

For the purposes of TTCN, the production in Recommendations X.680 which states:

SubtypeElements ::= SingleValue | ConstrainedSubtype | ValueRange | PermittedAlphabet | SizeConstraint | TypeConstraint | InnerTypeConstraint

is redefined to be

SubtypeElements ::= SingleValue | ConstrainedSubtype | ValueRange | PermittedAlphabet | SizeConstraint | TypeConstraint | InnerTypeConstraint | ASN1_Encoding

This means that ASN1_Encoding can be applied anywhere that a TypeConstraint can be applied: to the whole of an ASN1_Type or any ASN.1 Type within the ASN1_Type or to a SET OF or SEQUENCE OF type (by placing the ASN1_Encoding in parentheses immediately after the keyword SET or SEQUENCE – unlike for a SizeConstraint in such a position, the parentheses are required since there is no backwards compatibility argument for allowing their omission).

For the purpose of TTCN, the following productions in Recommendation X.680:

BuiltinValue ::=

BitStringType |
BooleanType |
CharacterStringType |
ChoiceType |
ChoiceType |
EmbeddedPDUType |
EnumeratedType |
ExternalType |
InstanceOfType |
IntegerType |

NullType |
ObjectClassFieldType |
OctetStringType |
RealType |
SequenceOfType |
SetType |
SetOfType |
TaggedType

ReferencedType ::=
DefinedType |
UsefulType |
SelectionType |
TypeFromObject |
ValueSetFromObjects

DefinedType ::=
Externaltypereference |
Typereference |
ParameterizedType|
ParameterizedValueSetType

Elements ::=
SubtypeElements |
ObjectSetElements |
"("ElementSetSpec")"

are redefined to be

BuiltinValue ::=
BitStringType |
BooleanType |
CharacterStringType |
ChoiceType |
EmbeddedPDUType |
EnumeratedType |
ExternalType |
IntegerType |
NullType |
ObjectIdentifierType |
RealType |
SequenceOfType |
SequenceOfType |
SetType |
SetOfType |
TaggedType

ReferencedType ::=
DefinedType |
UsefulType |
SelectionType

DefinedType ::=
Externaltypereference |
Typereference

Elements ::=
SubtypeElements |
"("ElementSetSpec")"*/

/*STATIC SEMANTICS – Each terminal type reference used within the Type production shall be one of the following: ASN1_LocalType typereference, TS_TypeIdentifier or PDU_Identifier. */

/* STATIC SEMANTICS – ASN.1 type definitions used within TTCN shall not use external type references as defined in Recommendation X.680. */

123 ASN1_LocalType ::= Typeassignment

/* REFERENCE – Where Typeassignment is a non-terminal defined in Recommendation X.680. */

/* STATIC SEMANTICS – ASN.1 type definitions used within TTCN shall not use external type references as defined in Recommendation X.680. */

A.3.3.10.6 ASN.1 Type Definitions by Reference

124 ASN1_TypeRefsOrGroup ::= ASN1_TypeRefs | ASN1_TypeRefsGroup
125 ASN1_TypeRefsGroup ::= **$ASN1_TypeRefsGroup** ASN1_TypeRefsGroupId {ASN1_TypeRefsOrGroup}+ **$End_ASN1_TypeRefsGroup**
126 ASN1_TypeRefsGroupId ::= **$ASN1_TypeRefsGroupId** ASN1_TypeGroupIdentifier
127 ASN1_TypeRefs ::= **$Begin_ASN1_TypeRefs** [ASN1_TypeRefsGroupRef] {[CollComment] ASN1_TypeRef}+ [Comment] **$End_ASN1_TypeRefs**
/* NOTE – Collective comments may be used in this table according to Figure 2. */
128 ASN1_TypeRefsGroupRef ::= **$ASN1_TypeRefsGroupRef** ASN1_TypeGroupReference
129 ASN1_TypeRef ::= **$ASN1_TypeRef** ASN1_TypeId ASN1_TypeReference ASN1_ModuleId [EncVariationId] [Comment] **$End_ASN1_TypeRef**
/* STATIC SEMANTICS – ASN1_TypeId shall not be specified with a FullIdentifier. */
130 ASN1_TypeReference ::= **$ASN1_TypeReference** TypeReference
131 TypeReference ::**=** typereference
/* REFERENCE – Where typereference is a non-terminal defined in Recommendation X.680. */
/* STATIC SEMANTICS – If the ASN.1 type definition has a reference to another type in the same ASN.1 Module, the referenced type is implicitly imported (in the same way as for a TTCN module). */
132 ASN1_ModuleId ::= **$ASN1_ModuleId** ASN1_ModuleIdentifier
133 ASN1_ModuleIdentifier ::**=** ModuleIdentifier
/* REFERENCE – Where ModuleIdentifier is a non-terminal defined in Recommendation X.680. */
/* STATIC SEMANTICS – ModuleIdentifier shall be unique within the domain of interest. */

A.3.3.10.7 Test Suite Operation Definitions

134 TS_OpDefs ::= **$TS_OpDefs** {TS_OpDefOrGroup}+ **$End_TS_OpDefs**
135 TS_OpDefOrGroup ::= TS_OpDef | TS_OpDefGroup
136 TS_OpDefGroup ::= **$TS_OpDefGroup** TS_OpDefGroupId {TS_OpDefOrGroup}+ **$End_TS_OpDefGroup**
137 TS_OpDefGroupId ::= **$TS_OpDefGroupId** TS_OpDefGroupIdentifier
138 TS_OpDefGroupIdentifier ::= Identifier
139 TS_OpDef ::= **$Begin_TS_OpDef** TS_OpId [TS_OpGroupRef] TS_OpResult [Comment] TS_OpDescription [Comment] **$End_TS_OpDef**
140 TS_OpId ::= **$TS_OpId** TS_OpId&ParList
141 TS_OpId&ParList ::= TS_OpIdentifier [FormalParList]
/* STATIC SEMANTICS – A Test Suite Operation formal parameter Type shall be a PredefinedType, TS_TypeIdentifier, PDU_Identifier or ASP_Identifier, or the meta-type **PDU**. */
142 TS_OpIdentifier ::= Identifier
143 TS_OpGroupRef ::= **$TS_OpGroupRef** TS_OpGroupReference
144 TS_OpGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TS_OpGroupIdentifier "/"}
145 TS_OpGroupIdentifier ::= Identifier
146 TS_OpResult ::= **$TS_OpResult** TypeOrPDU
/* STATIC SEMANTICS – TypeOrPDU shall be a PredefinedType, TS_TypeIdentifier, PDU_Identifier or ASP_Identifier, or the meta-type **PDU**. */
147 TS_OpDescription ::= **$TS_OpDescription** BoundedFreeText

A.3.3.10.8 Test Suite Operation Procedural Definitions

148 TS_ProcDefs ::= **$TS_ProcDefs** {TS_ProcDefOrGroup}+ **$End_TS_ProcDefs**
149 TS_ProcDefOrGroup ::= TS_ProcDef | TS_ProcDefGroup
150 TS_ProcDefGroup ::= **$TS_ProcDefGroup** TS_ProcDefGroupId {TS_ProcDefOrGroup}+ **$End_TS_ProcDefGroup**
151 TS_ProcDefGroupId ::= **$TS_ProcDefGroupId** TS_ProcDefGroupIdentifier
152 TS_ProcDefGroupIdentifier ::= Identifier
153 TS_ProcDef ::= **$Begin_TS_ProcDef** TS_ProcId [TS_ProcGroupRef] TS_ProcResult [Comment] TS_ProcDescription [Comment] **$End_TS_ProcDef**
/* LEXICAL REQUIREMENT – Comments may be embedded within TS_ProcDescription by enclosing them within "/*" and "*/" but may not be nested. They may be carried within TTCN.MP but shall be removed before parsing the TTCN.MP. */
154 TS_ProcId ::= **$TS_ProcId** TS_ProcId&ParList
155 TS_ProcId&ParList ::= TS_ProcIdentifier [FormalParList]
/* STATIC SEMANTICS – A procedural Test Suite Operation formal parameter Type shall be a PredefinedType, TS_TypeIdentifier, PDU_Identifier or ASP_Identifier, or the meta-type **PDU**. */
156 TS_ProcIdentifier ::= Identifier
157 TS_ProcGroupRef ::= **$TS_ProcGroupRef** TS_ProcGroupReference
158 TS_ProcGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TS_ProcGroupIdentifier "/"}
159 TS_ProcGroupIdentifier ::= Identifier

160 TS_ProcResult ::= **$TS_ProcResult** TypeOrPDU
/* STATIC SEMANTICS – TypeOrPDU shall be a PredefinedType, TS_TypeIdentifier, PDU_Identifier or ASP_Identifier, or the meta-type **PDU**. */
161 TS_ProcDescription ::= **$TS_ProcDescription** TS_OpProcDef **$End_TS_ProcDescription**
162 TS_OpProcDef ::= [VarBlock] ProcStatement
/* NOTE – Comments are allowed within TS_OpProcDef, starting with "/*" and ending with "*/", but it is assumed that these comments are removed before the syntax is parsed. Hence the BNF does not include the syntax of such embedded comments. */
163 VarBlock ::= **VAR** VarDcls **ENDVAR**
164 VarDcls ::= {VarDcl SemiColon}
165 VarDcl ::= [**STATIC**] VarIdentifiers Colon TypeOrPDU [Colon Value]
166 VarIdentifiers ::= VarIdentifier {Comma VarIdentifier}
167 VarIdentifier ::= Identifier
168 ProcStatement ::= ReturnValueStatement | Assignment | IfStatement | WhileLoop | CaseStatement | ProcBlock
169 ReturnValueStatement ::= **RETURNVALUE** Expression
170 IfStatement ::= **IF** Expression **THEN** {ProcStatement SemiColon}+ [**ELSE** {ProcStatement SemiColon}+] **ENDIF**
171 WhileLoop ::= **WHILE** Expression **DO** {ProcStatement SemiColon}+ **ENDWHILE**
172 CaseStatement ::= **CASE** Expression **OF** {CaseClause SemiColon}+ [**ELSE** {ProcStatement SemiColon}+] **ENDCASE**
173 CaseClause ::= IntegerLabel Colon ProcStatement
174 IntegerLabel ::= Number | TS_ParIdentifier | TS_ConstIdentifier
175 ProcBlock ::= **BEGIN** {ProcStatement SemiColon}+ **END**

A.3.3.11 Parameterization and Selection

A.3.3.11.1 General

176 Parameterization&Selection ::= [TS_ParDclsOrGroup] [SelectExprDefsOrGroup]

A.3.3.11.2 Test Suite Parameter Declarations

177 TS_ParDclsOrGroup ::= TS_ParDcls | TS_ParDclsGroup
178 TS_ParDclsGroup ::= **$TS_ParDclsGroup** TS_ParDclsGroupId {TS_ParDclsOrGroup}+ **$End_TS_ParDclsGroup**
179 TS_ParDclsGroupId ::= **$TS_ParDclsGroupId** TS_ParDclsGroupIdentifier
180 TS_ParDclsGroupIdentifier ::= Identifier
181 TS_ParDcls ::= **$Begin_TS_ParDcls** [TS_ParGroupRef] {[CollComment] TS_ParDcl}+ [Comment] **$End_TS_ParDcls**
/* NOTE – Collective comments may be used in this table according to Figure 2. */
182 TS_ParGroupRef ::= **$TS_ParGroupRef** TS_ParGroupReference
183 TS_ParGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TS_ParGroupIdentifier "/"}
184 TS_ParGroupIdentifier ::= Identifier
185 TS_ParDcl ::= **$TS_ParDcl** TS_ParId TS_ParType [TS_ParDefault] PICS_PIXITref [Comment] **$End_TS_ParDcl**
186 TS_ParId ::= **$TS_ParId** TS_ParIdentifier
187 TS_ParIdentifier ::= Identifier
188 TS_ParType ::= **$TS_ParType** TypeOrPDU
/* STATIC SEMANTICS – TypeOrPDU shall be a PredefinedType, TS_TypeIdentifier, PDU_Identifier or ASP_Identifier, or the meta-type **PDU**. */
189 TS_ParDefault ::= **$TS_ParDefault** [DefaultValue]
190 DefaultValue ::= Expression
/* STATIC SEMANTICS – DefaultValue shall not contain TS_Variables or TC_Variables and shall resolve to a constant value. */
/* OPERATIONAL SEMANTICS – DefaultValue shall evaluate to an element of its declared type. */
191 PICS_PIXITref ::= **$PICS_PIXITref** BoundedFreeText

A.3.3.11.3 Test Case Selection Expression Definitions

192 SelectExprDefsOrGroup ::= SelectExprDefs | SelectExprDefsGroup
193 SelectExprDefsGroup ::= **$SelectExprDefsGroup** SelectExprDefsGroupId {SelectExprDefsOrGroup}+ **$End_SelectExprDefsGroup**
194 SelectExprDefsGroupId ::= **$SelectExprDefsGroupId** SelectExprDefsGroupIdentifier
195 SelectExprDefsGroupIdentifier ::= Identifier
196 SelectExprDefs ::= **$Begin_SelectExprDefs** [SelectExprGroupRef] {[CollComment] SelectExprDef}+ [Comment] **$End_SelectExprDefs**
/* NOTE – Collective comments may be used in this table according to Figure 2. */
197 SelectExprGroupRef ::= **$SelectExprGroupRef** SelectExprGroupReference
198 SelectExprGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {SelectExprGroupIdentifier "/"}
199 SelectExprGroupIdentifier ::= Identifier
200 SelectExprDef ::= **$SelectExprDef** SelectExprId SelectExpr [Comment] **$End_SelectExprDef**
201 SelectExprId ::= **$SelectExprId** SelectExprIdentifier

202 SelectExprIdentifier ::= Identifier

203 SelectExpr ::= **$SelectExpr** SelectionExpression

204 SelectionExpression ::= Expression

/* STATIC SEMANTICS – SelectionExpression shall only contain LiteralValues, TS_ParIdentifiers, TS_ConstIdentifiers and SelectExprIdentifiers*/

/* OPERATIONAL SEMANTICS – SelectionExpression shall evaluate to a specific BOOLEAN value. */

/* STATIC SEMANTICS – Expression shall not recursively refer (neither directly nor indirectly) to the SelExprIdentifier being defined by that Expression. */

A.3.3.12 Declarations

A.3.3.12.1 General

205 Declarations ::= [TS_ConstDclsOrGroup] [TS_ConstRefsOrGroup] [TS_VarDclsOrGroup] [TC_VarDclsOrGroup] [PCO_TypeDclsOrGroup] [PCO_DclsOrGroup] [CP_DclsOrGroup] [TimerDclsOrGroup] [TCompDclsOrGroup TCompConfigDcls]

/* STATIC SEMANTICS – PCOs shall be optional. */

A.3.3.12.2 Test Suite Constant Declarations

206 TS_ConstDclsOrGroup ::= TS_ConstDcls | TS_ConstDclsGroup

207 TS_ConstDclsGroup ::= **$TS_ConstDclsGroup** TS_ConstDclsGroupId {TS_ConstDclsOrGroup}+ **$End_TS_ConstDclsGroup**

208 TS_ConstDclsGroupId ::= **$TS_ConstDclsGroupId** TS_ConstDclsGroupIdentifier

209 TS_ConstDclsGroupIdentifier ::= Identifier

210 TS_ConstDcls ::= **$Begin_TS_ConstDcls** [TS_ConstGroupRef] {[CollComment] TS_ConstDcl}+ [Comment] **$End_TS_ConstDcls**

/* NOTE – Collective comments may be used in this table according to Figure 2. */

211 TS_ConstGroupRef ::= **$TS_ConstGroupRef** TS_ConstGroupReference

212 TS_ConstGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TS_ConstGroupIdentifier "/"}

213 TS_ConstGroupIdentifier ::= Identifier

214 TS_ConstDcl ::= **$TS_ConstDcl** TS_ConstId TS_ConstType TS_ConstValue [Comment] **$End_TS_ConstDcl**

215 TS_ConstId ::= **$TS_ConstId** TS_ConstIdentifier

216 TS_ConstIdentifier ::= Identifier

217 TS_ConstType ::= **$TS_ConstType** Type

/* STATIC SEMANTICS – Type shall not be a structured type, PDU type, ASP type or CM type expressed in tabular form. */

218 TS_ConstValue ::= **$TS_ConstValue** DeclarationValue

219 DeclarationValue ::= Expression

/* STATIC SEMANTICS – DeclarationValue shall not contain TS_Variables or TC_Variables and shall resolve to a constant value. */

/* OPERATIONAL SEMANTICS – DeclarationValue shall evaluate to an element of its declared type. */

A.3.3.12.3 Test Suite Constant Declarations by Reference

220 TS_ConstRefsOrGroup ::= TS_ConstRefs | TS_ConstRefsGroup

221 TS_ConstRefsGroup ::= **$TS_ConstRefsGroup** TS_ConstRefsGroupId {TS_ConstRefsOrGroup}+ **$End_TS_ConstRefsGroup**

222 TS_ConstRefsGroupId ::= **$TS_ConstRefsGroupId** TS_ConstRefsGroupIdentifier

223 TS_ConstRefsGroupIdentifier ::= Identifier

224 TS_ConstRefs ::= **$Begin_TS_ConstRefs** [TS_ConstRefsGroupRef] {[CollComment] TS_ConstRef}+ [Comment] **$End_TS_ConstRefs**

/* NOTE – Collective comments may be used in this table according to Figure 2. */

225 TS_ConstRefsGroupRef ::= **$TS_ConstRefsGroupRef** TS_ConstGroupReference

226 TS_ConstRef ::= **$TS_ConstRef** TS_ConstId TS_ConstType ASN1_ValueReference ASN1_ModuleId [Comment] **$End_TS_ConstRef**

/* STATIC SEMANTICS – Type in TS_ConstType shall be either a PredefinedType or an ASN1_Type imported by an ASN.1 Type Definition By Reference from the module referenced by ASN1_ModuleId. */

227 ASN1_ValueReference ::= **$ASN1_ValueReference** ValueReference

228 ValueReference ::= valuereference

/* REFERENCE – valuereference is a non-terminal defined in Recommendation X.680. */

/* STATIC SEMANTICS – The value shall correspond to an element of the type in TS_ConstType. */

A.3.3.12.4 Test Suite Variable Declarations

TS_VarDclsOrGroup ::= TS_VarDcls | TS_VarDclsGroup
TS_VarDclsGroup ::= **$TS_VarDclsGroup** TS_VarDclsGroupId {TS_VarDclsOrGroup}+ **$End_TS_VarDclsGroup**
TS_VarDclsGroupId ::= **$TS_VarDclsGroupId** TS_VarDclsGroupIdentifier
TS_VarDclsGroupIdentifier ::= Identifier
TS_VarDcls ::= **$Begin_TS_VarDcls** [TS_VarGroupRef] {[CollComment] TS_VarDcl}+ [Comment] **$End_TS_VarDcls**
/* NOTE – Collective comments may be used in this table according to Figure 2. */
TS_VarGroupRef ::= **$TS_VarGroupRef** TS_VarGroupReference
TS_VarGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TS"VarGroupIdentifier "/"}
TS_VarGroupIdentifier ::= Identifier
TS_VarDcl ::= **$TS_VarDcl** TS_VarId TS_VarType TS_VarValue [Comment] **$End_TS_VarDcl**
TS_VarId ::= **$TS_VarId** TS_VarIdentifier
TS_VarIdentifier ::= Identifier
TS_VarType ::= **$TS_VarType** TypeOrPDU
/* STATIC SEMANTICS – TypeOrPDU shall be a PredefinedType, TS_TypeIdentifier, PDU_Identifier or ASP_Identifier, or the meta-type **PDU**. */
TS_VarValue ::= **$TS_VarValue** [DeclarationValue]

A.3.3.12.5 Test Case Variable Declarations

TC_VarDclsOrGroup ::= TC_VarDcls | TC_VarDclsGroup
TC_VarDclsGroup ::= **$TC_VarDclsGroup** TC_VarDclsGroupId {TC_VarDclsOrGroup}+ **$End_TC_VarDclsGroup**
TC_VarDclsGroupId ::= **$TC_VarDclsGroupId** TC_VarDclsGroupIdentifier
TC_VarDclsGroupIdentifier ::= Identifier
TC_VarDcls ::= **$Begin_TC_VarDcls** [TC_VarGroupRef] {[CollComment] TC_VarDcl}+ [Comment] **$End_TC_VarDcls**
/* NOTE – Collective comments may be used in this table according to Figure 2. */
TC_VarGroupRef ::= **$TC_VarGroupRef** TC_VarGroupReference
TC_VarGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TC_VarGroupIdentifier "/"}
TC_VarGroupIdentifier ::= Identifier
TC_VarDcl ::= **$TC_VarDcl** TC_VarId TC_VarType TC_VarValue [Comment] **$End_TC_VarDcl**
TC_VarId ::= **$TC_VarId** TC_VarIdentifier
TC_VarIdentifier ::= Identifier
TC_VarType ::= **$TC_VarType** TypeOrPDU
/* STATIC SEMANTICS – TypeOrPDU shall be a PredefinedType, TS_TypeIdentifier, PDU_Identifier or ASP_Identifier, or the meta-type **PDU**. */
TC_VarValue ::= **$TC_VarValue** [DeclarationValue]

A.3.3.12.6 PCO Type Declarations

PCO_TypeDclsOrGroup ::= PCO_TypeDcls | PCO_TypeDclsGroup
PCO_TypeDclsGroup ::= **$PCO_TypeDclsGroup** PCO_TypeDclsGroupId {PCO_TypeDclsOrGroup}+ **$End_PCO_TypeDclsGroup**
PCO_TypeDclsGroupId ::= **$PCO_TypeDclsGroupId** PCO_TypeDclsGroupIdentifier
PCO_TypeDclsGroupIdentifier ::= Identifier
PCO_TypeDcls ::= **$Begin_PCO_TypeDcls** [PCO_TypeGroupRef] {[CollComment] PCO_TypeDcl}+ [Comment] **$End_PCO_TypeDcls**
/* NOTE – Collective comments may be used in this table according to Figure 2. */
PCO_TypeGroupRef ::= **$PCO_TypeGroupRef** PCO_GroupReference
PCO_TypeDcl ::= **$PCO_TypeDcl** PCO_TypeId RoleOrComment **$End_PCO_TypeDcl**
PCO_TypeId ::= **$PCO_TypeId** PCO_TypeIdentifier
PCO_TypeIdentifier ::= Identifier
RoleOrComment ::= P_Role [Comment] | Comment
/* NOTE – Since each PCO_Type in a PCO Type Declaration Table has to have a role specified in either the Role or Comment column, at least one of P_Role or Comment is required to be present. */

A.3.3.12.7 PCO Declarations

PCO_DclsOrGroup ::= PCO_Dcls | PCO_DclsGroup
PCO_DclsGroup ::= **$PCO_DclsGroup** PCO_DclsGroupId {PCO_DclsOrGroup}+ **$End_PCO_DclsGroup**
PCO_DclsGroupId ::= **$PCO_DclsGroupId** PCO_DclsGroupIdentifier
PCO_DclsGroupIdentifier ::= Identifier

269 PCO_Dcls ::= **$Begin_PCO_Dcls** [PCO_GroupRef] {[CollComment] PCO_Dcl}+ [Comment] **$End_PCO_Dcls**
/* NOTE – Collective comments may be used in this table according to Figure 2. */
/* STATIC SEMANTICS – To be in accordance with Recommendation X.290, the number of PCOs shall relate to the test method used. */
270 PCO_GroupRef ::= **$PCO_GroupRef** PCO_GroupReference
271 PCO_GroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {PCO_GroupIdentifier "/"}
272 PCO_GroupIdentifier ::= Identifier
273 PCO_Dcl ::= **$PCO_Dcl** PCO_Id PCO_TypeId&MuxValue [P_Role] [Comment] **$End_PCO_Dcl**
274 PCO_Id ::= **$PCO_Id** PCO_Identifier
275 PCO_Identifier ::= Identifier
276 PCO_TypeId&MuxValue ::= **$PCO_TypeId** PCO_TypeIdentifier ["(" MuxValue ")"]
277 MuxValue ::= TS_ParIdentifier
278 P_Role ::= **$PCO_Role** [PCO_Role]
279 PCO_Role **::= UT** | **LT**

A.3.3.12.8 CP Declarations

280 CP_DclsOrGroup ::= CP_Dcls | CP_DclsGroup
281 CP_DclsGroup ::= **$CP_DclsGroup** CP_DclsGroupId {CP_DclsOrGroup}+ **$End_CP_DclsGroup**
282 CP_DclsGroupId ::= **$CP_DclsGroupId** CP_DclsGroupIdentifier
283 CP_DclsGroupIdentifier ::= Identifier
284 CP_Dcls ::= **$Begin_CP_Dcls** [CP_GroupRef] {[CollComment] CP_Dcl}+ [Comment] **$End_CP_Dcls**
/* NOTE – Collective comments may be used in this table according to Figure 2. */
285 CP_GroupRef ::= **$CP_GroupRef** CP_GroupReference
286 CP_GroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {CP_GroupIdentifier "/"}
287 CP_GroupIdentifier ::= Identifier
288 CP_Dcl ::= **$CP_Dcl** CP_Id [Comment] **$End_CP_Dcl**
289 CP_Id ::= **$CP_Id** CP_Identifier
290 CP_Identifier ::= Identifier

A.3.3.12.9 Timer Declarations

291 TimerDclsOrGroup ::= TimerDcls | TimerDclsGroup
292 TimerDclsGroup ::= **$TimerDclsGroup** TimerDclsGroupId {TimerDclsOrGroup}+ **$End_TimerDclsGroup**
293 TimerDclsGroupId ::= **$TimerDclsGroupId** TimerDclsGroupIdentifier
294 TimerDclsGroupIdentifier ::= Identifier
295 TimerDcls ::= **$Begin_TimerDcls** [TimerGroupRef] {[CollComment] TimerDcl}+ [Comment] **$End_TimerDcls**
/* NOTE – Collective comments may be used in this table according to Figure 2. */
296 TimerGroupRef ::= **$TimerGroupRef** TimerGroupReference
297 TimerGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TimerGroupIdentifier "/"}
298 TimerGroupIdentifier ::= Identifier
299 TimerDcl ::= **$TimerDcl** TimerId Duration Unit [Comment] **$End_TimerDcl**
300 TimerId ::= **$TimerId** TimerIdentifier
301 TimerIdentifier ::= Identifier
302 Duration ::= **$Duration** [DeclarationValue]
/* OPERATIONAL SEMANTICS – DeclarationValue shall evaluate to a non-zero positive INTEGER. */
303 Unit ::= **$Unit** TimeUnit
304 TimeUnit ::= **ps** | **ns** | **us** | **ms** | **s** | **min**
/* STATIC SEMANTICS – If a timer is derived from the PICS/PIXIT, then the timer declaration shall specify the same units as the PICS/PIXIT entry. */

A.3.3.12.10 Test Component Declarations

305 TCompDclsOrGroup ::= TCompDcls | TCompDclsGroup
306 TCompDclsGroup ::= **$TCompDclsGroup** TCompDclsGroupId {TCompDclsOrGroup}+ **$End_TCompDclsGroup**
307 TCompDclsGroupId ::= **$TCompDclsGroupId** TCompDclsGroupIdentifier
308 TCompDclsGroupIdentifier ::= Identifier
309 TCompDcls ::= **$Begin_TCompDcls** [TCompGroupRef] {[CollComment] TCompDcl}+ [Comment] **$End_TCompDcls**
/* NOTE – Collective comments may be used in this table according to Figure 2. */
310 TCompGroupRef ::= **$TCompGroupRef** TCompGroupReference
311 TCompGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TCompGroupIdentifier "/"}
312 TCompGroupIdentifier ::= Identifier
313 TCompDcl ::= **$TCompDcl** TCompId C_Role NumOf_PCOs NumOf_CPs [Comment] **$End_TCompDcl**
314 TCompId ::= **$TCompId** TCompIdentifier
315 TCompIdentifier::= Identifier

316 C_Role ::= **$TCompRole** TCompRole
317 TCompRole **::= MTC** | **PTC**
318 NumOf_PCOs ::= **$NumOf_PCOs** Num_PCOs
319 Num_PCOs ::= Number
320 NumOf_CPs ::= **$NumOf_CPs** Num_CPs
321 Num_CPs ::= Number

A.3.3.12.11 Test Component Configuration Declarations

322 TCompConfigDcls ::= **$TCompConfigDcls** {TCompConfigDclOrGroup}+ **$End_TCompConfigDcls**
323 TCompConfigDclOrGroup ::= TCompConfigDcl | TCompConfigDclGroup
324 TCompConfigDclGroup ::= **$TCompConfigDclGroup** TCompConfigDclGroupId {TCompConfigDclOrGroup}+ **$End_TCompConfigDclGroup**
325 TCompConfigDclGroupId ::= **$TCompConfigDclGroupId** TCompConfigDclGroupIdentifier
326 TCompConfigDclGroupIdentifier ::= Identifier
327 TCompConfigDcl ::= **$Begin_TCompConfigDcl** TCompConfigId [TCompConfigGroupRef] [Comment] TCompConfigInfos [Comment] **$End_TCompConfigDcl**
328 TCompConfigId ::= **$TCompConfigId** TCompConfigIdentifier
329 TCompConfigIdentifier ::= Identifier
330 TCompConfigGroupRef ::= **$TCompConfigGroupRef** TCompConfigGroupReference
331 TCompConfigGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {TCompConfigGroupIdentifier "/"}
332 TCompConfigGroupIdentifier ::= Identifier
333 TCompConfigInfos ::= **$TCompConfigInfos** {TCompConfigInfo}+ **$End_TCompConfigInfos**
/* STATIC SEMANTICS – Exactly one of the TCompConfigInfos shall be for a Test Component which has a TCompRole which is **MTC**. */
334 TCompConfigInfo ::= **$TCompConfigInfo** TCompUsed PCOs_Used CPs_Used [Comment] **$End_TCompConfigInfo**
335 TCompUsed ::= **$TCompUsed** TCompIdentifier
336 PCOs_Used ::= **$PCOs_Used** [PCO_List]
337 PCO_List ::= PCO_Identifier {Comma PCO_Identifier}
/* STATIC SEMANTICS – The number of PCOs in the PCO_List shall be the same as in the Test Component declaration. */
/* STATIC SEMANTICS – A given PCO_Identifier shall not be used more than once in the same Test Component Configuration. */
338 CPs_Used ::= **$CPs_Used** [CP_List]
339 CP_List ::= CP_Identifier {Comma CP_Identifier}
/* STATIC SEMANTICS – For a PTC, the number of CPs in the CP_List shall be the same as in the Test Component declaration. */
/* STATIC SEMANTICS – For an MTC, the number of CPs in the CP_List shall be no more than the number in the Test Component declaration. */
/* STATIC SEMANTICS – A given CP_Identifier shall not appear more than once in a given CP_List. */
/* STATIC SEMANTICS – Each CP_Identifier which is used in a Test Component Configuration shall appear in the CP_List of precisely two Test Components in that Configuration. */

A.3.3.13 ASP, PDU and CM Type Definitions

A.3.3.13.1 General

340 ComplexDefinitions ::= [ASP_TypeDefs] [PDU_TypeDefs] [CM_TypeDefs] [AliasDefsOrGroup]
/* STATIC SEMANTICS – PDUs shall be optional. */

A.3.3.13.2 ASP Type Definitions

341 ASP_TypeDefs ::= **$ASP_TypeDefs** [TTCN_ASP_TypeDefs] [ASN1_ASP_TypeDefs] [ASN1_ASP_TypeDefsByRefOrGroup] **$End_ASP_TypeDefs**

A.3.3.13.3 Tabular ASP Type Definitions

342 TTCN_ASP_TypeDefs ::= **$TTCN_ASP_TypeDefs** {TTCN_ASP_TypeDefOrGroup}+ **$End_TTCN_ASP_TypeDefs**
343 TTCN_ASP_TypeDefOrGroup ::= TTCN_ASP_TypeDef | TTCN_ASP_TypeDefGroup
344 TTCN_ASP_TypeDefGroup ::= **$TTCN_ASP_TypeDefGroup** TTCN_ASP_TypeDefGroupId {TTCN_ASP_TypeDefOrGroup}+ **$End_TTCN_ASP_TypeDefGroup**
345 TTCN_ASP_TypeDefGroupId ::= **$TTCN_ASP_TypeDefGroupId** ASP_GroupIdentifier
346 TTCN_ASP_TypeDef ::= **$Begin_TTCN_ASP_TypeDef** ASP_Id [ASP_GroupRef] PCO_Type [Comment] [ASP_ParDcls] [Comment] **$End_TTCN_ASP_TypeDef**
347 ASP_Id ::= **$ASP_Id** ASP_Id&FullId
348 ASP_Id&FullId ::= ASP_Identifier [FullIdentifier]

349 ASP_Identifier ::= Identifier
/* STATIC SEMANTICS – Identifier may be AliasIdentifier provided that it is being used in the behaviour column of a behaviour table (i.e. in a Behaviour Description). */
350 ASP_GroupRef ::= **$ASP_GroupRef** ASP_GroupReference
351 ASP_GroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ASP_GroupIdentifier "/"}
352 ASP_GroupIdentifier ::= Identifier
353 PCO_Type ::= **$PCO_Type** [PCO_TypeIdentifier]
/* STATIC SEMANTICS – If there is no PCO_Type declaration table, then PCO_TypeIdentifier shall be one of the PCO types used in the PCO declaration table. */
/* STATIC SEMANTICS – If only a single PCO is defined within a test suite, then PCO_TypeIdentifier is optional. */
354 ASP_ParDcls ::= **$ASP_ParDcls** {ASP_ParDcl} **$End_ASP_ParDcls**
355 ASP_ParDcl ::= **$ASP_ParDcl** ASP_ParId ASP_ParType [Comment] **$End_ASP_ParDcl**
356 ASP_ParId ::= **$ASP_ParId** ASP_ParIdOrMacro
357 ASP_ParIdOrMacro ::= ASP_ParId&FullId | MacroSymbol
/* STATIC SEMANTICS – The MacroSymbol shall be used only in combination with a reference to a Structured Type. */
358 ASP_ParId&FullId ::= ASP_ParIdentifier [FullIdentifier]
359 ASP_ParIdentifier ::= Identifier
360 ASP_ParType ::= **$ASP_ParType** Type&Attributes
/* STATIC SEMANTICS – Type shall be a PredefinedType or TS_TypeIdentifier, PDU_Identifier, or **PDU**. */

A.3.3.13.4 ASN.1 ASP Type Definitions

361 ASN1_ASP_TypeDefs ::= **$ASN1_ASP_TypeDefs** {ASN1_ASP_TypeDefOrGroup} **$End_ASN1_ASP_TypeDefs**
362 ASN1_ASP_TypeDefOrGroup ::= ASN1_ASP_TypeDef | ASN1_ASP_TypeDefGroup
363 ASN1_ASP_TypeDefGroup ::= **$ASN1_ASP_TypeDefGroup** ASN1_ASP_TypeDefGroupId {ASN1_ASP_TypeDefOrGroup}+ **$End_ASN1_ASP_TypeDefGroup**
364 ASN1_ASP_TypeDefGroupId ::= **$ASN1_ASP_TypeDefGroupId** ASN1_ASP_GroupIdentifier
365 ASN1_ASP_TypeDef ::= **$Begin_ASN1_ASP_TypeDef** ASP_Id [ASN1_ASP_GroupDef] PCO_Type [Comment] [ASN1_TypeDefinition] [Comment] **$End_ASN1_ASP_TypeDef**
366 ASN1_ASP_GroupRef ::= **$ASN1_ASP_GroupRef** ASN1_ASP_GroupReference
367 ASN1_ASP_GroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ASN1_ASP_GroupIdentifier "/"}
368 ASN1_ASP_GroupIdentifier ::= Identifier

A.3.3.13.5 ASN.1 ASP Type Definitions by Reference

369 ASN1_ASP_TypeDefsByRefOrGroup ::= ASN1_ASP_TypeDefsByRef | ASN1_ASP_TypeDefsByRefGroup
370 ASN1_ASP_TypeDefsByRefGroup ::= **$ASN1_ASP_TypeDefsByRefGroup** ASN1_ASP_TypeDefsByRefGroupId {ASN1_ASP_TypeDefsByRefOrGroup}+ **$End_ASN1_ASP_TypeDefsByRefGroup**
371 ASN1_ASP_TypeDefsByRefGroupId ::= **$ASN1_ASP_TypeDefsByRefGroupId** ASN1_ASP_GroupIdentifier
372 ASN1_ASP_TypeDefsByRef ::= **$Begin_ASN1_ASP_TypeDefsByRef** [ASN1_ASP_DefsByRefGroupRef] {[CollComment] ASN1_ASP_TypeDefByRef}+ [Comment] **$End_ASN1_ASP_TypeDefsByRef**
/* NOTE – Collective comments may be used in this table according to Figure 2. */
373 ASN_ASP_DefsByRefGroupRef ::= **$ASN1_ASP_DefsByRefGroupRef** ASN1_ASP_GroupReference
374 ASN1_ASP_TypeDefByRef ::= **$ASN1_ASP_TypeDefByRef** ASP_Id PCO_Type ASN1_TypeReference ASN1_ModuleId [Comment] **$End_ASN1_ASP_TypeDefByRef**
/* STATIC SEMANTICS – ASP_Id shall not be specified with a FullIdentifier. */

A.3.3.13.6 PDU Type Definitions

375 PDU_TypeDefs ::= **$PDU_TypeDefs** [TTCN_PDU_TypeDefs] [ASN1_PDU_TypeDefs] [ASN1_PDU_TypeDefsByRefOrGroup] **$End_PDU_TypeDefs**

A.3.3.13.7 Tabular PDU Type Definitions

376 TTCN_PDU_TypeDefs ::= **$TTCN_PDU_TypeDefs** {TTCN_PDU_TypeDefOrGroup}+ **$End_TTCN_PDU_TypeDefs**
377 TTCN_PDU_TypeDefOrGroup ::= TTCN_PDU_TypeDef | TTCN_PDU_TypeDefGroup
378 TTCN_PDU_TypeDefGroup ::= **$TTCN_PDU_TypeDefGroup** TTCN_PDU_TypeDefGroupId {TTCN_PDU_TypeDefOrGroup}+ **$End_TTCN_PDU_TypeDefGroup**
379 TTCN_PDU_TypeDefGroupId ::= **$TTCN_PDU_TypeDefGroupId** PDU_GroupIdentifier
380 TTCN_PDU_TypeDef ::= **$Begin_TTCN_PDU_TypeDef** PDU_Id [PDU_GroupRef] PCO_Type [PDU_EncodingId] [EncVariationId] [Comment] [PDU_FieldDcls] [Comment] **$End_TTCN_PDU_TypeDef**
/* STATIC SEMANTICS – If a PDU is sent or received only embedded in ASPs within the whole test suite, then PCO_TypeIdentifier (in PCO_Type) is optional. */
381 PDU_Id ::= **$PDU_Id** PDU_Id&FullId
382 PDU_Id&FullId ::= PDU_Identifier [FullIdentifier]

383 PDU_Identifier ::= Identifier

/* STATIC SEMANTICS – Identifier may be AliasIdentifier provided that it is being used in the behaviour column of a behaviour table (i.e. in a Behaviour Description). */

384 PDU_GroupRef ::= **$PDU_GroupRef** PDU_GroupReference

385 PDU_GroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {PDU_GroupIdentifier "/"}

386 PDU_GroupIdentifier ::= Identifier

387 PDU_EncodingId ::= **$PDU_EncodingId** [EncodingRuleIdentifier]

388 PDU_FieldDcls ::= **$PDU_FieldDcls** {PDU_FieldDcl} **$End_PDU_FieldDcls**

389 PDU_FieldDcl ::= **$PDU_FieldDcl** PDU_FieldId PDU_FieldType [PDU_FieldEncoding] [Comment] **$End_PDU_FieldDcl**

390 PDU_FieldId ::= **$PDU_FieldId** PDU_FieldIdOrMacro

391 PDU_FieldIdOrMacro ::= PDU_FieldId&FullId | MacroSymbol

/* STATIC SEMANTICS – The MacroSymbol shall be used only in combination with a reference to a Structured Type. */

392 MacroSymbol ::= "<-"

393 PDU_FieldId&FullId ::= PDU_FieldIdentifier [FullIdentifier]

394 PDU_FieldIdentifier ::= Identifier

395 PDU_FieldType ::= **$PDU_FieldType** Type&Attributes

/* STATIC SEMANTICS – Type shall be a PredefinedType or TS_TypeIdentifier, PDU_Identifier, or **PDU**. */

396 Type&Attributes ::= (Type [LengthAttribute]) | **PDU**

/* OPERATIONAL SEMANTICS – The set of values defined by LengthAttribute shall be a true subset of the values of the base type. */

/* STATIC SEMANTICS – LengthAttribute shall be provided only when the base type is a string type (i.e. BITSTRING, HEXSTRING, OCTETSTRING or CharacterString) or derived from a string type. */

397 LengthAttribute ::= SingleLength | RangeLength

398 SingleLength ::= "[" Bound "]"

399 Bound ::= Number | TS_ParIdentifier | TS_ConstIdentifier

/* OPERATIONAL SEMANTICS – Bound shall evaluate to a non-negative INTEGER value or INFINITY. */

400 RangeLength ::= "[" LowerBound To UpperBound "]"

/* OPERATIONAL SEMANTICS – LowerBound shall be less than UpperBound. */

401 LowerBound ::= Bound

402 UpperBound ::= Bound | **INFINITY**

A.3.3.13.8 ASN.1 PDU Type Definitions

403 ASN1_PDU_TypeDefs ::= **$ASN1_PDU_TypeDefs** {ASN1_PDU_TypeDefOrGroup} **$End_ASN1_PDU_TypeDefs**

404 ASN1_PDU_TypeDefOrGroup ::= ASN1_PDU_TypeDef | ASN1_PDU_TypeDefGroup

405 ASN1_PDU_TypeDefGroup ::= **$ASN1_PDU_TypeDefGroup** ASN1_PDU_TypeDefGroupId {ASN1_PDU_TypeDefOrGroup}+ **$End_ASN1_PDU_TypeDefGroup**

406 ASN1_PDU_TypeDefGroupId ::= **$ASN1_PDU_TypeDefGroupId** ASN1_PDU_GroupIdentifier

407 ASN1_PDU_TypeDef ::= **$Begin_ASN1_PDU_TypeDef** PDU_Id [ASN1_PDU_GroupRef] PCO_Type [PDU_EncodingId] [EncVariationId] [Comment] [ASN1_TypeDefinition] [Comment] **$End_ASN1_PDU_TypeDef**

/* STATIC SEMANTICS – If a PDU is sent or received only embedded in ASPs within the whole test suite, then PCO_TypeIdentifier (in PCO_Type) is optional. */

408 ASN1_PDU_GroupRef ::= **$ASN1_PDU_GroupRef** ASN1_PDU_GroupReference

409 ASN1_PDU_GroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ASN1_PDU_GroupIdentifier "/"}

410 ASN1_PDU_GroupIdentifier ::= Identifier

A.3.3.13.9 ASN.1 PDU Type Definitions by Reference

411 ASN1_PDU_TypeDefsByRefOrGroup ::= ASN1_PDU_TypeDefsByRef | ASN1_PDU_TypeDefsByRefGroup

412 ASN1_PDU_TypeDefsByRefGroup ::= **$ASN1_PDU_TypeDefsByRefGroup** ASN1_PDU_TypeDefsByRefGroupId {ASN1_PDU_TypeDefsByRefOrGroup}+ **$End_ASN1_PDU_TypeDefsByRefGroup**

413 ASN1_PDU_TypeDefsByRefGroupId ::= **$ASN1_PDU_TypeDefsByRefGroupId** ASN1_PDU_GroupIdentifier

414 ASN1_PDU_TypeDefsByRef ::= **$Begin_ASN1_PDU_TypeDefsByRef** [ASN1_PDU_DefsByRefGroupRef] {[CollComment] ASN1_PDU_TypeDefByRef}+ [Comment] **$End_ASN1_PDU_TypeDefsByRef**

/* NOTE – Collective comments may be used in this table according to Figure 2. */

415 ASN1_PDU_DefsByRefGroupRef ::= **$ASN1_PDU_DefsByRefGroupRef** ASN1_PDU_GroupReference

416 ASN1_PDU_TypeDefByRef ::= **$ASN1_PDU_TypeDefByRef** PDU_Id PCO_Type ASN1_TypeReference ASN1_ModuleId [PDU_EncodingId] [EncVariationId] [Comment] **$End_ASN1_PDU_TypeDefByRef**

/* STATIC SEMANTICS – If a PDU is sent or received only embedded in ASPs within the whole test suite, then PCO_TypeIdentifier (in PCO_Type) is optional. */

/* STATIC SEMANTICS – PDU_Id shall not be specified with a FullIdentifier. */

A.3.3.13.10 CM Type Definitions

417 CM_TypeDefs ::= **$CM_TypeDefs** [TTCN_CM_TypeDefs] [ASN1_CM_TypeDefs] **$End_CM_TypeDefs**

A.3.3.13.11 Tabular CM Type Definition

418 TTCN_CM_TypeDefs ::= **$TTCN_CM_TypeDefs** {TTCN_CM_TypeDefOrGroup}+ **$End_TTCN_CM_TypeDefs**
419 TTCN_CM_TypeDefOrGroup ::= TTCN_CM_TypeDef | TTCN_CM_TypeDefGroup
420 TTCN_CM_TypeDefGroup ::= **$TTCN_CM_TypeDefGroup** TTCN_CM_TypeDefGroupId
{TTCN_CM_TypeDefOrGroup}+ **$End_TTCN_CM_TypeDefGroup**
421 TTCN_CM_TypeDefGroupId ::= **$TTCN_CM_TypeDefGroupId** CM_GroupIdentifier
422 TTCN_CM_TypeDef ::= **$Begin_TTCN_CM_TypeDef** CM_Id [CM_GroupRef] [Comment] [CM_ParDcls] [Comment]
$End_TTCN_CM_TypeDef
423 CM_Id ::= **$CM_Id** CM_Identifier
424 CM_Identifier ::= Identifier
425 CM_GroupRef ::= **$CM_GroupRef** CM_GroupReference
426 CM_GroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {CM_GroupIdentifier "/"}
427 CM_GroupIdentifier ::= Identifier
428 CM_ParDcls ::= **$CM_ParDcls** {CM_ParDcl} **$End_CM_ParDcls**
429 CM_ParDcl ::= **$CM_ParDcl** CM_ParId CM_ParType [Comment] **$End_CM_ParDcl**
430 CM_ParId ::= **$CM_ParId** CM_ParIdOrMacro
431 CM_ParIdOrMacro ::= CM_ParIdentifier | MacroSymbol
/* STATIC SEMANTICS – The MacroSymbol shall be used only in combination with a reference to a Structured Type. */
432 CM_ParIdentifier ::= Identifier
433 CM_ParType ::= **$CM_ParType** Type&Attributes

A.3.3.13.12 ASN.1 CM Type Definitions

434 ASN1_CM_TypeDefs ::= **$ASN1_CM_TypeDefs** {ASN1_CM_TypeDefOrGroup}+ **$End_ASN1_CM_TypeDefs**
435 ASN1_CM_TypeDefOrGroup ::= ASN1_CM_TypeDef | ASN1_CM_TypeDefGroup
436 ASN1_CM_TypeDefGroup ::= **$ASN1_CM_TypeDefGroup** ASN1_CM_TypeDefGroupId
{ASN1_CM_TypeDefOrGroup}+ **$End_ASN1_CM_TypeDefGroup**
437 ASN1_CM_TypeDefGroupId ::= **$ASN1_CM_TypeDefGroupId** ASN1_CM_GroupIdentifier
438 ASN1_CM_TypeDef ::= **$Begin_ASN1_CM_TypeDef** CM_Id [ASN1_CM_GroupRef] [Comment]
[ASN1_TypeDefinition] [Comment] **$End_ASN1_CM_TypeDef**
439 ASN1_CM_GroupRef ::= **$ASN1_CM_GroupRef** ASN1_CM_GroupReference
440 ASN1_CM_GroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ASN1_CM_GroupIdentifier "/"}
441 ASN1_CM_GroupIdentifier ::= Identifier

A.3.3.13.13 Varieties of Encoding Definition

442 EncodingDefs ::= **$EncodingDefs** [EncodingDefinitionsOrGroup] [EncodingVariations] [InvalidFieldEncodingDefs]
$End_EncodingDefs

A.3.3.13.13.1 Encoding Definitions

443 EncodingDefinitionsOrGroup ::= EncodingDefinitions | EncodingDefinitionsGroup
444 EncodingDefinitionsGroup ::= **$EncodingDefinitionsGroup** EncodingDefinitionsGroupId
{EncodingDefinitionsOrGroup}+ **$End_EncodingDefinitionsGroup**
445 EncodingDefinitionsGroupId ::= **$EncodingDefinitionsGroupId** EncodingGroupIdentifier
446 EncodingDefinitions ::= **$Begin_EncodingDefinitions** [EncodingGroupRef] {[CollComment] EncodingDefinition}+
[Comment] **$End_EncodingDefinitions**
/* NOTE – Collective comments may be used in this table according to Figure 2. */
447 EncodingGroupRef ::= **$EncodingGroupRef** EncodingGroupReference
448 EncodingGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {EncodingGroupIdentifier "/"}
449 EncodingGroupIdentifier ::= Identifier
450 EncodingDefinition ::= **$EncodingDefinition** EncodingRuleId EncodingRef EncodingDefault [Comment]
$End_EncodingDefinition
/* OPERATIONAL SEMANTICS – No more than one EncodingRuleIdentifier shall have an EncodingDefault containing a DefaultExpression which evaluates to TRUE. */
451 EncodingRuleId ::= **$EncodingRuleId** EncodingRuleIdentifier
452 EncodingRuleIdentifier ::= Identifier
453 EncodingRef ::= **$EncodingRef** EncodingReference
454 EncodingReference ::= BoundedFreeText
455 EncodingDefault ::= **$EncodingDefault** [DefaultExpression]
456 DefaultExpression ::= Expression
/* STATIC SEMANTICS – DefaultExpression shall only contain LiteralValues, TS_ParIdentifiers and TS_ConstIdentifiers. */

A.3.3.13.13.2 Encoding Variations

457 EncodingVariations ::= **$EncodingVariations** {EncodingVariationSetOrGroup}+ **$End_EncodingVariations**
458 EncodingVariationSetOrGroup ::= EncodingVariationSet | EncodingVariationSetGroup
459 EncodingVariationSetGroup ::= **$EncodingVariationSetGroup** EncodingVariationSetGroupId {EncodingVariationSetOrGroup}+ **$End_EncodingVariationSetGroup**
460 EncodingVariationSetGroupId ::= **$EncodingVariationSetGroupId** EncVariationGroupIdentifier
461 EncodingVariationSet ::= **$Begin_EncodingVariationSet** EncodingRuleId [EncVariationGroupRef] Encoding_TypeList [Comment] EncodingVariationList [Comment] **$End_EncodingVariationSet**
462 EncVariationGroupRef ::= **$EncVariationGroupRef** EncVariationGroupReference
463 EncVariationGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {EncVariationGroupIdentifier "/"}
464 EncVariationGroupIdentifier ::= Identifier
465 EncodingVariationList ::= **$EncodingVariationList** {EncodingVariation}+ **$End_EncodingVariationList**
466 Encoding_TypeList ::= **$Encoding_TypeList** [TypeList]
467 TypeList ::=Type {Comma Type}
/* STATIC SEMANTICS – Type shall not be an ASP_Identifier, PDU_Identifier or StructIdentifier, since such types may be encoded by encoding rules but not by field encodings. */
468 EncodingVariation ::= **$EncodingVariation** EncodingVariationId VariationRef VariationDefault [Comment] **$End_EncodingVariation**
/* OPERATIONAL SEMANTICS – No more than one EncodingIdentifier shall have a VariationDefault containing a DefaultExpression which evaluates to TRUE. */
469 EncodingVariationId ::= **$EncodingVariationId** EncVariationId&ParList
470 EncVariationId&ParList ::= EncVariationIdentifier [FormalParList]
471 EncVariationIdentifier ::= Identifier
472 VariationRef ::= **$VariationRef** VariationReference
473 VariationReference ::= BoundedFreeText
474 VariationDefault ::= **$VariationDefault** [DefaultExpression]

A.3.3.13.13.3 Invalid Encoding Definitions

475 InvalidFieldEncodingDefs ::= **$InvalidFieldEncodingDefs** {InvalidFieldEncodingDefOrGroup}+ **$End_InvalidFieldEncodingDefs**
476 InvalidFieldEncodingDefOrGroup ::= InvalidFieldEncodingDef | InvalidFieldEncodingGroup
477 InvalidFieldEncodingGroup ::= **$InvalidFieldEncodingGroup** InvalidFieldEncodingGroupId {InvalidFieldEncodingOrGroup}+ **$End_InvalidFieldEncodingGroup**
478 InvalidFieldEncodingGroupId ::= **$InvalidFieldEncodingGroupId** InvalidFieldEncodingGroupIdentifier
479 InvalidFieldEncodingDef ::= **$Begin_InvalidFieldEncodingDef** InvalidFieldEncodingId [InvalidFieldEncodingGroupRef] Encoding_TypeList [Comment] InvalidFieldEncodingDefinition [Comment] **$End_InvalidFieldEncodingDef**
480 InvalidFieldEncodingId ::= **$InvalidFieldEncodingId** InvalidFieldEncodingId&ParList
481 InvalidFieldEncodingId&ParList ::= InvalidFieldEncodingIdentifier [FormalParList]
482 InvalidFieldEncodingIdentifier ::= Identifier
483 InvalidFieldEncodingGroupRef ::= **$InvalidFieldEncodingGroupRef** InvalidFieldEncodingGroupReference
484 InvalidFieldEncodingGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {InvalidFieldEncodingGroupIdentifier "/"}
485 InvalidFieldEncodingGroupIdentifier ::= Identifier
486 InvalidFieldEncodingDefinition ::= **$InvalidFieldEncodingDefinition** TS_OpProcDef **$End_InvalidFieldEncodingDefinition**
/* OPERATIONAL SEMANTICS – TS_OpProcDef shall produce a BitString result, to be interpreted as the encoding to be transmitted high order bit first. */

A.3.3.13.14 Alias Definitions

487 AliasDefsOrGroup ::= AliasDefs | AliasDefsGroup
488 AliasDefsGroup ::= **$AliasDefsGroup** AliasDefsGroupId {AliasDefsOrGroup}+ **$End_AliasDefsGroup**
489 AliasDefsGroupId ::= **$AliasDefsGroupId** AliasDefsGroupIdentifier
490 AliasDefsGroupIdentifier ::= Identifier
491 AliasDefs ::= **$Begin_AliasDefs** [AliasGroupRef] {[CollComment] AliasDef}+ [Comment] **$End_AliasDefs**
/* NOTE – Collective comments may be used in this table according to Figure 2. */
492 AliasGroupRef ::= **$AliasGroupRef** AliasGroupReference
493 AliasGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {AliasGroupIdentifier "/"}
494 AliasGroupIdentifier ::= Identifier
495 AliasDef ::= **$AliasDef** AliasId ExpandedId [Comment] **$End_AliasDef**
496 AliasId ::= **$AliasId** AliasIdentifier

497 AliasIdentifier ::= Identifier

/* STATIC SEMANTICS – An AliasIdentifier shall be used only in a statement line of a behaviour description. */

/* STATIC SEMANTICS – An AliasIdentifier shall be used only where an ASP_Identifier or PDU_Identifier is valid. */

498 ExpandedId ::= **$ExpandedId** Expansion

499 Expansion ::= ASP_Identifier | PDU_Identifier

A.3.3.14 Constraints Part

500 ConstraintsPart ::= **$ConstraintsPart** [TS_TypeConstraints] [ASP_Constraints] [PDU_Constraints] [CM_Constraints] **$End_ConstraintsPart**

A.3.3.15 Test Suite Type Constraint Declarations

501 TS_TypeConstraints ::= **$TS_TypeConstraints** [StructTypeConstraints] [ASN1_TypeConstraints] **$End_TS_TypeConstraints**

A.3.3.16 Structured Type Constraint Declarations

502 StructTypeConstraints ::= **$StructTypeConstraints** {StructTypeConstraintOrGroup}+ **$End_StructTypeConstraints**

503 StructTypeConstraintOrGroup ::= StructTypeConstraint | StructTypeConstraintGroup

504 StructTypeConstraintGroup ::= **$StructTypeConstraintGroup** StructTypeConstraintGroupId {StructTypeConstraintOrGroup}+ **$End_StructTypeConstraintGroup**

505 StructTypeConstraintGroupId ::= **$StructTypeConstraintGroupId** StructTypeConstraintGroupIdentifier

506 StructTypeConstraint ::= **$Begin_StructTypeConstraint** ConsId [StructTypeConstraintGroupRef] StructId DerivPath [EncVariationId] [Comment] ElemValues [Comment] **$End_StructTypeConstraint**

/* STATIC SEMANTICS – The FullIdentifier that is part of Struct_Id shall not be used. */

/* STATIC SEMANTICS – A modified constraint shall have the same parameter list as its base constraint. In particular, there shall be no parameters omitted from or added to this list. */

507 StructTypeConstraintGroupRef ::= **$StructTypeConstraintGroupRef** StructTypeConstraintGroupReference

508 StructTypeConstraintGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {StructTypeConstraintGroupIdentifier "/"}

509 StructTypeConstraintGroupIdentifier ::= Identifier

510 EncVariationId ::= **$EncVariationId** [EncVariationCall]

511 EncVariationCall ::= EncVariationIdentifier [ActualParList]

512 ElemValues ::= **$ElemValues** {ElemValue}+ **$End_ElemValues**

513 ElemValue ::= **$ElemValue** ElemId ConsValue [PDU_FieldEncoding] [Comment] **$End_ElemValue**

/* STATIC SEMANTICS – The FullIdentifier that is part of ElemId shall not be used. */

/* STATIC SEMANTICS – Parameterized Element values in a base constraint shall not be modified or explicitly omitted in a modified constraint. */

514 PDU_FieldEncoding ::= **$PDU_FieldEncoding** [PDU_FieldEncodingCall]

515 PDU_FieldEncodingCall ::= EncVariationCall | InvalidFieldEncodingCall

516 InvalidFieldEncodingCall ::= InvalidFieldEncodingIdentifier (ActualParList | "(" ")")

A.3.3.17 ASN.1 Type Constraint Declarations

517 ASN1_TypeConstraints ::= **$ASN1_TypeConstraints** {ASN1_TypeConstraintOrGroup}+ **$End_ASN1_TypeConstraints**

518 ASN1_TypeConstraintOrGroup ::= ASN1_TypeConstraint | ASN1_TypeConstraintGroup

519 ASN1_TypeConstraintGroup ::= **$ASN1_TypeConstraintGroup** ASN1_TypeConstraintGroupId {ASN1_TypeConstraintOrGroup}+ **$End_ASN1_TypeConstraintGroup**

520 ASN1_TypeConstraintGroupId ::= **$ASN1_TypeConstraintGroupId** ASN1_TypeConstraintGroupIdentifier

521 ASN1_TypeConstraint ::= **$Begin_ASN1_TypeConstraint** ConsId [ASN1_TypeConstraintGroupRef] ASN1_TypeId DerivPath [EncVariationId] [Comment] ASN1_ConsValue [Comment] **$End_ASN1_TypeConstraint**

/* STATIC SEMANTICS – The FullIdentifier that is part of ASN1_TypeId shall not be used. */

/* STATIC SEMANTICS – A modified constraint shall have the same parameter list as its base constraint. In particular, there shall be no parameters omitted from or added to this list. */

522 ASN1_TypeConstraintGroupRef ::= **$ASN1_TypeConstraintGroupRef** ASN1_TypeConstraintGroupReference

523 ASN1_TypeConstraintGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ASN1_TypeConstraintGroupIdentifier "/"}

524 ASN1_TypeConstraintGroupIdentifier ::= Identifier

A.3.3.18 ASP Constraint Declarations

525 ASP_Constraints ::= **$ASP_Constraints** [TTCN_ASP_Constraints] [ASN1_ASP_Constraints] **$End_ASP_Constraints**

A.3.3.19 Tabular ASP Constraint Declarations

526 TTCN_ASP_Constraints ::= **$TTCN_ASP_Constraints** {TTCN_ASP_ConstraintOrGroup}+ **$End_TTCN_ASP_Constraints**

527 TTCN_ASP_ConstraintOrGroup ::= TTCN_ASP_Constraint | TTCN_ASP_ConstraintGroup

528 TTCN_ASP_ConstraintGroup ::= **$TTCN_ASP_ConstraintGroup** TTCN_ASP_ConstraintGroupId {TTCN_ASP_ConstraintOrGroup}+ **$End_TTCN_ASP_ConstraintGroup**

529 TTCN_ASP_ConstraintGroupId ::= **$TTCN_ASP_ConstraintGroupId** ASP_ConstraintGroupIdentifier

530 TTCN_ASP_Constraint ::= **$Begin_TTCN_ASP_Constraint** ConsId [ASP_ConstraintGroupRef] ASP_Id DerivPath [Comment] [ASP_ParValues] [Comment] **$End_TTCN_ASP_Constraint**

/* STATIC SEMANTICS – The FullIdentifier that is part of ASP_Id shall not be used. */

/* STATIC SEMANTICS – If an ASP is substructured, then the constraints for ASPs of that type shall have the same structure. */

/* STATIC SEMANTICS – A modified constraint shall have the same parameter list as its base constraint. In particular, there shall be no parameters omitted from or added to this list. */

531 ASP_ConstraintGroupRef ::= **$ASP_ConstraintGroupRef** ASP_ConstraintGroupReference

532 ASP_ConstraintGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ASP_ConstraintGroupIdentifier "/"}

533 ASP_ConstraintGroupIdentifier ::= Identifier

534 ASP_ParValues ::= **$ASP_ParValues** {ASP_ParValue} **$End_ASP_ParValues**

535 ASP_ParValue ::= **$ASP_ParValue** ASP_ParId ConsValue [Comment] **$End_ASP_ParValue**

/* STATIC SEMANTICS – The FullIdentifier that is part of ASP_ParId shall not be used. */

/* STATIC SEMANTICS – If an ASP definition refers to a Structured Type as a substructure of a parameter (i.e., with a parameter name), then the corresponding constraint shall have the same parameter name in the corresponding position in the parameter name column of the constraint and the value shall be a reference to a constraint for that parameter (i.e., for that substructure in accordance with the definition of the Structured Type). */

/* STATIC SEMANTICS – If an ASP definition refers to a parameter specified as being of metatype PDU, then in a corresponding constraint, the value for that parameter shall be specified as the name of a PDU constraint, or formal parameter. */

/* STATIC SEMANTICS – Use of structured constraints by macro expansion in a constraint shall not be used unless the corresponding ASP definition also references the same Structured Type by macro expansion. */

/* STATIC SEMANTICS – Parameterized ASP parameter values in a base constraint shall not be modified or explicitly omitted in a modified constraint. */

A.3.3.20 ASN.1 ASP Constraint Declarations

536 ASN1_ASP_Constraints ::= **$ASN1_ASP_Constraints** {ASN1_ASP_ConstraintOrGroup}+ **$End_ASN1_ASP_Constraints**

537 ASN1_ASP_ConstraintOrGroup ::= ASN1_ASP_Constraint | ASN1_ASP_ConstraintGroup

538 ASN1_ASP_ConstraintGroup ::= **$ASN1_ASP_ConstraintGroup** ASN1_ASP_ConstraintGroupId {ASN1_ASP_ConstraintOrGroup}+ **$End_ASN1_ASP_ConstraintGroup**

539 ASN1_ASP_ConstraintGroupId ::= **$ASN1_ASP_ConstraintGroupId** ASN1_ASP_ConstraintGroupIdentifier

540 ASN1_ASP_Constraint ::= **$Begin_ASN1_ASP_Constraint** ConsId [ASN1_ASP_ConstraintGroupRef] ASP_Id DerivPath [Comment] [ASN1_ConsValue] [Comment] **$End_ASN1_ASP_Constraint**

/* STATIC SEMANTICS – The FullIdentifier that is part of ASP_Id shall not be used. */

/* STATIC SEMANTICS – If an ASP is substructured, then the constraints for ASPs of that type shall have a compatible ASN.1 structure (i.e., possibly with some groupings). */

/* STATIC SEMANTICS – A modified constraint shall have the same parameter list as its base constraint. In particular, there shall be no parameters omitted from or added to this list. */

541 ASN1_ASP_ConstraintGroupRef ::= **$ASN1_ASP_ConstraintGroupRef** ASN1_ASP_ConstraintGroupReference

542 ASN1_ASP_ConstraintGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ASN1_ASP_ConstraintGroupIdentifier "/"}

543 ASN1_ASP_ConstraintGroupIdentifier ::= Identifier

A.3.3.21 PDU Constraint Declarations

544 PDU_Constraints ::= **$PDU_Constraints** [TTCN_PDU_Constraints] [ASN1_PDU_Constraints] **$End_PDU_Constraints**

A.3.3.22 Tabular PDU Constraint Declarations

545 TTCN_PDU_Constraints ::= **$TTCN_PDU_Constraints** {TTCN_PDU_ConstraintOrGroup}+ **$End_TTCN_PDU_Constraints**

546 TTCN_PDU_ConstraintOrGroup ::= TTCN_PDU_Constraint | TTCN_PDU_ConstraintGroup

547 TTCN_PDU_ConstraintGroup ::= **$TTCN_PDU_ConstraintGroup** TTCN_PDU_ConstraintGroupId {TTCN_PDU_ConstraintOrGroup}+ **$End_TTCN_PDU_ConstraintGroup**

548 TTCN_PDU_ConstraintGroupId ::= **$TTCN_PDU_ConstraintGroupId** PDU_ConstraintGroupIdentifier

549 TTCN_PDU_Constraint ::= **$Begin_TTCN_PDU_Constraint** ConsId [PDU_ConstraintGroupRef] PDU_Id DerivPath [EncRuleId] [EncVariationId] [Comment] [PDU_FieldValues] [Comment] **$End_TTCN_PDU_Constraint**

/* STATIC SEMANTICS – The FullIdentifier that is part of PDU_Id shall not be used. */

/* STATIC SEMANTICS – If a PDU is substructured, then the constraints for PDUs of that type shall have the same structure*/

/* STATIC SEMANTICS – A modified constraint shall have the same parameter list as its base constraint. In particular, there shall be no parameters omitted from or added to this list. */

550 PDU_ConstraintGroupRef ::= **$PDU_ConstraintGroupRef** PDU_ConstraintGroupReference

551 PDU_ConstraintGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {PDU_ConstraintGroupIdentifier "/"}

552 PDU_ConstraintGroupIdentifier ::= Identifier

553 EncRuleId ::= **$EncRuleId** [EncodingRuleIdentifier]

554 ConsId ::= **$ConsId** ConsId&ParList

555 ConsId&ParList ::= ConstraintIdentifier [FormalParList]

556 ConstraintIdentifier ::= Identifier

557 DerivPath ::= **$DerivPath** [DerivationPath]

558 DerivationPath ::= {ConstraintIdentifier Dot}+

/* STATIC SEMANTICS – If a constraint definition is a modification of an existing constraint, the name of the constraint that is taken as the basis of this modification shall be referenced in the table in the derivation path entry. */

/* STATIC SEMANTICS – The first ConstraintIdentifier in DerivationPath shall be a base constraint identifier. */

/* STATIC SEMANTICS – The DerivationPath shall be the complete list of constraints in the order in which their modifications to the base constraint are to be applied. */

/* STATIC SEMANTICS – There shall be no white space between ConstraintIdentifier and Dot. */

559 PDU_FieldValues ::= **$PDU_FieldValues** {PDU_FieldValue} **$End_PDU_FieldValues**

560 PDU_FieldValue ::= **$PDU_FieldValue** PDU_FieldId ConsValue [PDU_FieldEncoding] [Comment] **$End_PDU_FieldValue**

/* STATIC SEMANTICS – The FullIdentifier that is part of PDU_FieldId shall not be used. */

/* STATIC SEMANTICS – If a PDU definition refers to a Structured Type as a substructure of a field (i.e., with a field name), then the corresponding constraint shall have the same field name in the corresponding position in the field name column of the constraint and the value shall be a reference to a constraint for that field (i.e., for that substructure in accordance with the definition of the Structured Type). */

/* STATIC SEMANTICS – If a PDU definition refers to a field specified as being of metatype PDU, then in a corresponding constraint, the value for that field shall be specified as the name of a PDU constraint, or formal parameter. */

/* STATIC SEMANTICS – Use of structured constraints by macro expansion in a constraint shall not be used unless the corresponding PDU definition also references the same Structured Type by macro expansion. */

/* STATIC SEMANTICS – Parameterized PDU field values in a base constraint shall not be modified or explicitly omitted in a modified constraint. */

561 ConsValue ::= **$ConsValue** ConstraintValue&Attributes

/* OPERATIONAL SEMANTICS – ConsValue shall evaluate to an element of the type specified for the ASP parameter, PDU field or structure element. This may include matching symbols compatible with the specified type. */

562 ConstraintValue&Attributes ::= ConstraintValue ValueAttributes

/* NOTE – ConstraintValue&Attributes can be reached via DefinedValue in the ASN.1 syntax. See the reference on the production 739 for Value. */

/* STATIC SEMANTICS – ConstraintValue shall fulfil all restrictions defined for the ASP parameter, PDU field or structure element type, including value ranges, value lists, alphabet restrictions and/or length restrictions and shall fulfil the restrictions defined by ValueAttributes. */

/* OPERATIONAL SEMANTICS – Any length specifications defined for the ASP parameter or PDU field type in the Test Suite Type declarations shall not conflict with the length specifications in the ASP or PDU type definition. */

/* STATIC SEMANTICS – Neither Test Suite Variables nor Test Case Variables shall be used in constraints, unless passed as actual parameters. In the latter case they shall be bound to a value and shall not be changed. */

563 ConstraintValue ::= ConstraintExpression | MatchingSymbol | ConsRef

/* STATIC SEMANTICS – When a ConstraintExpression is used in a Constraint, its terms shall not contain TS_VarIdentifier or TC_VarIdentifier. */

564 ConstraintExpression ::= Expression

/* OPERATIONAL SEMANTICS – ConstraintExpression shall evaluate to an element of the specified type. */

565 MatchingSymbol ::= Complement | Omit | AnyValue | AnyOrOmit | ValueList | ValueRange | SuperSet | SubSet | Permutation

/* NOTE – No matching symbol is considered to be a specific value. */

566 Complement ::= **COMPLEMENT** ValueList

567 Omit ::= Dash | **OMIT**

/* STATIC SEMANTICS – In ASN.1 constraints Omit shall be used only for ASP parameters or PDU fields that are declared OPTIONAL or DEFAULT. */

568 AnyValue ::= "?"

569 AnyOrOmit ::= "*"

570 ValueList ::= "(" ConstraintValue&Attributes {Comma ConstraintValue&Attributes} ")"

/* STATIC SEMANTICS – Each ConstraintValue&Attributes shall be of the type declared for the ASP parameter, PDU field, or structure element in which the ValueList is used. */

571 ValueRange ::= "(" ValRange ")"

/* STATIC SEMANTICS – ValueRange shall be used only on ASP parameter, PDU field, or structure element of type INTEGER. */

/* STATIC SEMANTICS – The set of values defined by ValueRange shall be a true subset of the values allowed by the ASP parameters, PDU fields or structure elements declared type. */

572 ValRange ::= (LowerRangeBound To UpperRangeBound)

/* OPERATIONAL SEMANTICS – LowerRangeBound shall be less than UpperRangeBound. */

573 LowerRangeBound ::= ConstraintExpression | Minus **INFINITY**

/* OPERATIONAL SEMANTICS – ConstraintExpression shall evaluate to a specific INTEGER value. */

574 UpperRangeBound ::= ConstraintExpression | **INFINITY**

/* OPERATIONAL SEMANTICS – ConstraintExpression shall evaluate to a specific INTEGER value. */

575 SuperSet ::= **SUPERSET** "(" ConstraintValue&Attributes ")"

/* STATIC SEMANTICS – The argument to SuperSet, i.e., ConstraintValue&Attributes, shall be of type SET OF. */

576 SubSet ::= **SUBSET** "(" ConstraintValue&Attributes ")"

/* STATIC SEMANTICS – The argument to SubSet, i.e., ConstraintValue&Attributes, shall be of type SET OF. */

577 Permutation ::= **PERMUTATION** ValueList

/* STATIC SEMANTICS – In ASN.1 constraints IF_PRESENT shall be used only for ASP parameters or PDU fields that are declared OPTIONAL /* STATIC SEMANTICS – The Permutation shall be used only inside a value of type SEQUENCE OF. */

/* STATIC SEMANTICS – The ValueList shall be of the type specified in the SEQUENCE OF. */

578 ValueAttributes ::= [ValueLength] [**IF_PRESENT**] [ASN1_Encoding]

/* STATIC SEMANTICS – In ASN.1 constraints IF_PRESENT shall be used only for ASP parameters or PDU fields that are declared OPTIONAL

or DEFAULT. */

/* STATIC SEMANTICS – ASN1_Encoding shall only be used for ValueAttributes in ASN.1 Type Constraints and ASN.1 PDU Constraints. */

579 ASN1_Encoding ::= **ENC** PDU_FieldEncodingCall

580 ValueLength ::= SingleValueLength | RangeValueLength

/* STATIC SEMANTICS – ValueLength shall be used only for ASP parameters, PDU fields or structure element that are declared as BITSTRING, HEXSTRING, OCTETSTRING, CharacterString, SEQUENCE OF or SET OF. */

/* STATIC SEMANTICS – ValueLength shall be used only in combination with the following mechanisms: Specificvalue, Complement, Omit, AnyValue, AnyOrOmit, AnyOrNone and Permutation. */

/* STATIC SEMANTICS – The set of values defined by ValueLength shall be a true subset of the values allowed by the ASP parameters, PDU fields or structure elements declared type. */

581 SingleValueLength ::= "[" ValueBound "]"

582 ValueBound ::= Number | TS_ParIdentifier | TS_ConstIdentifier | FormalParIdentifier

/* OPERATIONAL SEMANTICS – ValueBound shall evaluate to a specific non-negative INTEGER value. */

583 RangeValueLength ::= "[" LowerValueBound To UpperValueBound "]"

/* OPERATIONAL SEMANTICS – LowerValueBound shall be less than UpperValueBound. */

584 LowerValueBound ::= ValueBound

585 UpperValueBound ::= ValueBound | **INFINITY**

A.3.3.23 ASN.1 PDU Constraint Declarations

586 ASN1_PDU_Constraints ::= **$ASN1_PDU_Constraints** {ASN1_PDU_ConstraintOrGroup}+ **$End_ASN1_PDU_Constraints**

587 ASN1_PDU_ConstraintOrGroup ::= ASN1_PDU_Constraint | ASN1_PDU_ConstraintGroup

588 ASN1_PDU_ConstraintGroup ::= **$ASN1_PDU_ConstraintGroup** ASN1_PDU_ConstraintGroupId {ASN1_PDU_ConstraintOrGroup}+ **$End_ASN1_PDU_ConstraintGroup**

589 ASN1_PDU_ConstraintGroupId ::= **$ASN1_PDU_ConstraintGroupId** ASN1_PDU_ConstraintGroupIdentifier

590 ASN1_PDU_Constraint ::= **$Begin_ASN1_PDU_Constraint** ConsId [ASN1_PDU_ConstraintGroupRef] PDU_Id DerivPath [EncRuleId] [EncVariationId] [Comment] [ASN1_ConsValue] [Comment] **$End_ASN1_PDU_Constraint**

/* STATIC SEMANTICS – The FullIdentifier that is part of PDU_Id shall not be used. */

/* STATIC SEMANTICS – If a PDU is substructured, then the constraints for PDUs of that type shall have a compatible ASN.1 structure (i.e., possibly with some groupings). */

/* STATIC SEMANTICS – A modified constraint shall have the same parameter list as its base constraint. In particular, there shall be no parameters omitted from or added to this list. */

591 ASN1_PDU_ConstraintGroupRef ::= **$ASN1_PDU_ConstraintGroupRef** ASN1_PDU_ConstraintGroupReference

592 ASN1_PDU_ConstraintGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ASN1_PDU_ConstraintGroupIdentifier "/"}

593 ASN1_PDU_ConstraintGroupIdentifier ::= Identifier

594 ASN1_ConsValue ::= **$ASN1_ConsValue** ConstraintValue&AttributesOrReplace **$End_ASN1_ConsValue**

595 ConstraintValue&AttributesOrReplace ::= ConstraintValue&Attributes | Replacement {Comma Replacement}

596 Replacement ::= **REPLACE** ReferenceList **BY** ConstraintValue&Attributes | **OMIT** ReferenceList

/* STATIC SEMANTICS – Replacement shall be used only when DerivPath is specified. */

/* STATIC SEMANTICS – Parameterized replaced values in a base constraint shall not be modified or explicitly omitted in a modified constraint. */

597 ReferenceList ::= (ArrayRef | ComponentIdentifier | ComponentPosition) {ComponentReference}

A.3.3.24 CM Constraint Declarations

598 CM_Constraints ::= **$CM_Constraints** [TTCN_CM_Constraint**s**] [ASN1_CM_Constraint**s**] **$End_CM_Constraints**

A.3.3.25 Tabular CM Constraint Declarations

599 TTCN_CM_Constraints ::= **$TTCN_CM_Constraints** {TTCN_CM_ConstraintOrGroup}+ **$End_TTCN_CM_Constraints**

600 TTCN_CM_ConstraintOrGroup ::= TTCN_CM_Constraint | TTCN_CM_ConstraintGroup

601 TTCN_CM_ConstraintGroup ::= **$TTCN_CM_ConstraintGroup** TTCN_CM_ConstraintGroupId {TTCN_CM_ConstraintOrGroup}+ **$End_TTCN_CM_ConstraintGroup**

602 TTCN_CM_ConstraintGroupId ::= **$TTCN_CM_ConstraintGroupId** CM_ConstraintGroupIdentifier

603 TTCN_CM_Constraint ::= **$Begin_TTCN_CM_Constraint** ConsId [CM_ConstraintGroupRef] CM_Id DerivPath [Comment] [CM_ParValues] [Comment] **$End_TTCN_CM_Constraint**

604 CM_ConstraintGroupRef ::= **$CM_ConstraintGroupRef** CM_ConstraintGroupReference

605 CM_ConstraintGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {CM_ConstraintGroupIdentifier "/"}

606 CM_ConstraintGroupIdentifier ::= Identifier

607 CM_ParValues ::= **$CM_ParValues** {CM_ParValue} **$End_CM_ParValues**

608 CM_ParValue ::= **$CM_ParValue** CM_ParId ConsValue [Comment] **$End_CM_ParValue**

A.3.3.26 ASN.1 CM Constraint Declarations

609 ASN1_CM_Constraints ::= **$ASN1_CM_Constraints** {ASN1_CM_ConstraintOrGroup}+ **$End_ASN1_CM_Constraints**

610 ASN1_CM_ConstraintOrGroup ::= ASN1_CM_Constraint | ASN1_CM_ConstraintGroup

611 ASN1_CM_ConstraintGroup ::= **$ASN1_CM_ConstraintGroup** ASN1_CM_ConstraintGroupId {ASN1_CM_ConstraintOrGroup}+ **$End_ASN1_CM_ConstraintGroup**

612 ASN1_CM_ConstraintGroupId ::= **$ASN1_CM_ConstraintGroupId** ASN1_CM_ConstraintGroupIdentifier

613 ASN1_CM_Constraint ::= **$Begin_ASN1_CM_Constraint** ConsId [ASN1_CM_ConstraintGroupRef] CM_Id DerivPath [Comment] [ASN1_ConsValue] [Comment] **$End_ASN1_CM_Constraint**

614 ASN1_CM_ConstraintGroupRef ::= **$ASN1_CM_ConstraintGroupRef** ASN1_CM_ConstraintGroupReference

615 ASN1_CM_ConstraintGroupReference ::= [(SuiteIdentifier | TTCN_ModuleIdentifier) "/"] {ASN1_CM_ConstraintGroupIdentifier "/"}

616 ASN1_CM_ConstraintGroupIdentifier ::= Identifier

A.3.3.27 Dynamic Part

617 DynamicPart ::= **$DynamicPart** [TestCases] [TestStepLibrary] [DefaultsLibrary] **$End_DynamicPart**

A.3.3.28 Test Cases

618 TestCases ::= **$TestCases** {TestGroup | TestCase}+ **$End_TestCases**
619 TestGroup ::= **$TestGroup** TestGroupId {TestGroup | TestCase}+ **$End_TestGroup**
620 TestGroupId ::= **$TestGroupId** TestGroupIdentifier
621 TestGroupIdentifier ::= Identifier
622 TestCase ::= **$Begin_TestCase** TestCaseId TestGroupRef TestPurpose [Configuration] DefaultsRef [Comment] BehaviourDescription [Comment] **$End_TestCase**
623 TestCaseId ::= **$TestCaseId** TestCaseIdentifier
624 TestCaseIdentifier ::= Identifier
625 TestGroupRef ::= **$TestGroupRef** TestGroupReference
626 TestGroupReference ::= [SuiteIdentifier "/"] {TestGroupIdentifier "/"}
/* STATIC SEMANTICS – There shall be no white space on either side of the "/"s. */
627 TestPurpose ::= **$TestPurpose** BoundedFreeText
628 Configuration ::= **$Configuration** TCompConfigIdentifier
629 DefaultsRef::= **$DefaultsRef** [DefaultRefList]
630 DefaultRefList ::= DefaultReference {Comma DefaultReference}
631 DefaultReference ::= DefaultIdentifier [ActualParList]

A.3.3.29 Test Step Library

632 TestStepLibrary ::= **$TestStepLibrary** {TestStepGroup | TestStep}+ **$End_TestStepLibrary**
633 TestStepGroup ::= **$TestStepGroup** TestStepGroupId {TestStepGroup | TestStep}+ **$End_TestStepGroup**
634 TestStepGroupId ::= **$TestStepGroupId** TestStepGroupIdentifier
635 TestStepGroupIdentifier ::= Identifier
636 TestStep ::= **$Begin_TestStep** TestStepId TestStepRef Objective DefaultsRef [Comment] BehaviourDescription [Comment] **$End_TestStep**
637 TestStepId ::= **$TestStepId** TestStepId&ParList
638 TestStepId&ParList ::= TestStepIdentifier [FormalParList]
639 TestStepIdentifier ::= Identifier
640 TestStepRef ::= **$TestStepRef** TestStepGroupReference
641 TestStepGroupReference ::= [SuiteIdentifier "/"] {TestStepGroupIdentifier "/"}
/* STATIC SEMANTICS – There shall be no white space on either side of the "/"s. */
642 Objective ::= **$Objective** BoundedFreeText

A.3.3.30 Default Library

643 DefaultsLibrary ::= **$DefaultsLibrary** {DefaultGroup | Default}+ **$End_DefaultsLibrary**
644 DefaultGroup ::= **$DefaultGroup** DefaultGroupId {DefaultGroup | Default}+ **$End_DefaultGroup**
645 DefaultGroupId ::= **$DefaultGroupId** DefaultGroupIdentifier
646 Default ::= **$Begin_Default** DefaultId DefaultRef Objective [Comment] BehaviourDescription [Comment] **$End_Default**
/* STATIC SEMANTICS – BehaviourDescription shall not use tree attachment except for attaching local trees (i.e., Default behaviour trees shall not attach Test Steps). */
647 DefaultRef ::= **$DefaultRef** DefaultGroupReference
648 DefaultId ::= **$DefaultId** DefaultId&ParList
649 DefaultId&ParList ::= DefaultIdentifier [FormalParList]
650 DefaultIdentifier ::= Identifier
651 DefaultGroupReference ::= [SuiteIdentifier "/"] {DefaultGroupIdentifier "/"}
/* STATIC SEMANTICS – There shall be no white space on either side of the "/"s. */
652 DefaultGroupIdentifier ::= Identifier

A.3.3.31 Behaviour descriptions

653 BehaviourDescription ::= **$BehaviourDescription** RootTree {LocalTree} **$End_BehaviourDescription**
654 RootTree ::= {BehaviourLine}+
655 LocalTree ::= Header {BehaviourLine}+
656 Header ::= **$Header** TreeHeader
657 TreeHeader ::= TreeIdentifier [FormalParList]
658 TreeIdentifier ::= Identifier
659 FormalParList ::= "(" FormalPar&Type {SemiColon FormalPar&Type} ")"
660 FormalPar&Type ::= FormalParIdentifier {Comma FormalParIdentifier} Colon FormalParType
661 FormalParIdentifier ::= Identifier
662 FormalParType ::= Type | PCO_TypeIdentifier | **PDU** | **CP** | **TIMER**
/* STATIC SEMANTICS – In a Test Suite Operation or an encoding operation FormalParType shall not be a PCO type or the keyword CP*/
/* STATIC SEMANTICS – If a formal parameter is of type **PDU,** then that formal parameter shall not be used with a component reference (i.e. specific fields of the PDU cannot be referenced). */

A.3.3.32 Behaviour lines

663 BehaviourLine ::= **$BehaviourLine** LabelId Line Cref VerdictId [Comment] **$End_BehaviourLine**

664 Line ::= **$Line** Indentation StatementLine

665 Indentation ::= "[" Number "]"

/* STATIC SEMANTICS – Statements in the first level of alternatives in a behaviour description shall have the indentation value zero. */

/* STATIC SEMANTICS – Statements having a predecessor shall have the indentation value of the predecessor plus one as their indentation value. */

666 LabelId ::= **$LabelId** [Label]

667 Label ::= Identifier

668 Cref ::= **$Cref** [ConstraintReference]

669 ConstraintReference ::= ConsRef | FormalParIdentifier | AnyValue

/* STATIC SEMANTICS – ConsRef shall be present in conjunction with SEND, IMPLICIT SEND and RECEIVE and shall have a type which is consistent with (i.e. the same as or a subset of) the type of ASP, PDU or CM specified in the SEND, IMPLICIT_SEND or RECEIVE statement. A ConstraintReference is not needed for ASPs and CMs that have no parameters or PDUs that have no fields. It shall not be present with any other kind of TTCN statement. */

/* STATIC SEMANTICS – FormalParIdentifier shall resolve to a ConsRef. */

/* STATIC SEMANTICS – ConstraintReferences on SEND events shall not include any MatchingSymbol except Omit unless the MatchingSymbol is explicitly assigned specific values on the SEND event line. */

670 ConsRef ::= ConstraintIdentifier [ActualCrefParList]

671 ActualCrefParList ::= "(" ActualCrefPar {Comma ActualCrefPar} ")"

/* STATIC SEMANTICS – See static semantics on production 699. */

672 ActualCrefPar ::= Value

/* NOTE – Through Value, it is possible to reach MatchingSymbol, TS_ParIdentifier, TS_ConstIdentifier, TS_VarIdentifier, TC_VarIdentifier, FormalParIdentifier or ConsRef. */

673 VerdictId ::= **$VerdictId** [Verdict]

674 Verdict ::= Pass | Fail | Inconclusive | Result

/* STATIC SEMANTICS – Verdict shall not occur corresponding to entries in the behaviour tree which are any of the following: empty, an ATTACH construct, a REPEAT construct, a GOTO construct, an IMPLICIT SEND or a RETURN. */

675 Pass ::= **PASS** | **P** | "(" **PASS** ")" | "(" **P** ")"

676 Fail ::= **FAIL** | **F** | "(" **FAIL** ")" | "(" **F** ")"

677 Inconclusive ::= **INCONC** | **I** | "(" **INCONC** ")" | "(" **I** ")"

678 Result ::= **R**

/* STATIC SEMANTICS – R shall not be used on the LHS of an assignment. */

A.3.3.33 TTCN statements

679 StatementLine ::= (Event [Qualifier] [AssignmentList] [TimerOps]) | (Qualifier [AssignmentList] [TimerOps]) | (AssignmentList [TimerOps]) | TimerOps | Construct | ImplicitSend

680 Event ::= Send | Receive | Otherwise | Timeout | Done

/* STATIC SEMANTICS – A Receive, Otherwise or Timeout event shall only be followed by other Receive, Otherwise and Timeout events through the remainder of the set of alternatives in a fully expanded tree. As a consequence, Default trees will contain only Receive, Otherwise and Timeout events on the first level of alternatives. */

681 Qualifier ::= "[" Expression "]"

/* OPERATIONAL SEMANTICS – Qualifier shall evaluate to a specific BOOLEAN value. */

682 Send ::= [PCO_Identifier | CP_Identifier | FormalParIdentifier] "!" (ASP_Identifier | PDU_Identifier | CM_Identifier)

/* STATIC SEMANTICS – PCO_Identifier, CP_Identifier or FormalParIdentifier shall be present unless the test suite uses only one PCO and no CP. */

/* STATIC SEMANTICS – FormalParIdentifier shall resolve to a PCO_Identifier or CP_Identifier. */

/* STATIC SEMANTICS – Only CMs may be exchanged on CPs and only ASPs and PDUs may be exchanged on PCOs. */

683 ImplicitSend ::= "<" **IUT** "!" (ASP_Identifier | PDU_Identifier) ">"

/* STATIC SEMANTICS – ImplicitSend shall not be used unless the test method being used is one of the Remote Test Methods. */

684 Receive ::= [PCO_Identifier | CP_Identifier | FormalParIdentifier] "?" (ASP_Identifier | PDU_Identifier | CM_Identifier)

/* STATIC SEMANTICS – PCO_Identifier, CP_Identifier or FormalParIdentifier shall be present unless the test suite uses only one PCO and no CP. */

/* STATIC SEMANTICS – Only CMs may be exchanged on CPs and only ASPs and PDUs may be exchanged on PCOs. */

685 Otherwise ::= [PCO_Identifier | CP_Identifier | FormalParIdentifier] "?" **OTHERWISE**

/* STATIC SEMANTICS – PCO_Identifier, CP_Identifier or FormalParIdentifier shall be present unless the test suite uses only one PCO and no CP. */

/* STATIC SEMANTICS – FormalParIdentifier shall only be of PCO type or CP type. */

686 Timeout ::= "?" **TIMEOUT** [TimerIdentifier | FormalParIdentifier]

/* STATIC SEMANTICS – FormalParIdentifier shall only be of TIMER type. */

687 Done ::= "?" **DONE** "(" [TCompIdList] ")"

688 TCompIdList ::= TCompIdentifier {Comma TCompIdentifier}

689 Construct ::= GoTo | Attach | Repeat | Return | Activate | Create

690 Activate ::= **ACTIVATE** "(" [DefaultRefList] ")"

/* STATIC SEMANTICS – The ACTIVATE construct shall not be used in Default behaviour tables. */

691 Return ::= **RETURN**

/* STATIC SEMANTICS – The RETURN construct shall not be used except in Default behaviour trees (including any local trees within Default behaviour tables). */

692 Create ::= **CREATE** "(" CreateList ")"

693 CreateList ::= CreateTComp {Comma CreateTComp}

694 CreateTComp ::= TCompIdentifier Colon TreeReference [ActualParList]

/* STATIC SEMANTICS – TCompIdentifier shall not be of Role MTC. */

695 GoTo ::= ("->" | **GOTO**) Label

/* STATIC SEMANTICS – The label column shall contain labels referenced from the GoTo. */

/* STATIC SEMANTICS – Label shall be associated with the first of a set of alternatives, one of which is an ancestor node of the point from which the GoTo is to be made. */

/* STATIC SEMANTICS – GoTo shall be used only for jumps within one tree, i.e., within a Test Case root tree, a Test Step tree a Default tree and a local tree; and thus, each label used in a GoTo construct shall be found within the tree in which the GoTo is used. */

/* STATIC SEMANTICS – There shall be no ACTIVATE operation as an ancestor node of the GoTo construct on the branch of the tree between the Label and the GoTo. */

/* STATIC SEMANTICS – No GoTo shall be made to the first level of alternatives of local trees, Test Steps or Defaults. */

696 Attach ::= "+" TreeReference [ActualParList]

/* STATIC SEMANTICS – TreeReference shall not attach itself, either directly or indirectly, at its top level of indentation. */

/* STATIC SEMANTICS – The number of the actual parameters shall be the same as the number of the formal parameters. */

/* STATIC SEMANTICS – Formal and actual parameters of test steps shall be used in such a way that only valid TTCN is created by textual substitution. */

/* STATIC SEMANTICS – LiteralValue, TS_ParIdentifier, TS_ConstIdentifier, TS_VarIdentifier, TC_VarIdentifier, ConsRef, MatchingSymbol, FormalParIdentifier, PCO_Identifier and CP_Identifier may be passed as actual parameters to an attached tree. */

697 Repeat ::= **REPEAT** TreeReference [ActualParList] **UNTIL** Qualifier

/* STATIC SEMANTICS – TreeReference shall not attach itself, either directly or indirectly, at its top level of indentation. */

/* STATIC SEMANTICS – The number of the actual parameters shall be the same as the number of the formal parameters. */

/* STATIC SEMANTICS – LiteralValue, TS_ParIdentifier, TS_ConstIdentifier, TS_VarIdentifier, TC_VarIdentifier, ConsRef, MatchingSymbol, FormalParIdentifier, PCO_Identifier and CP_Identifier may be passed as actual parameters to the tree in a REPEAT statement. */

698 TreeReference ::= TestStepIdentifier | TreeIdentifier

/* STATIC SEMANTICS – TreeIdentifier shall be the name of one of the trees in the current behaviour description, i.e., local trees are not accessible outside the behaviour description in which they are specified. */

699 ActualParList ::= "(" ActualPar {Comma ActualPar} ")"

/* STATIC SEMANTICS – The number of the actual parameters shall be the same as the number of the formal parameters. */

/* OPERATIONAL SEMANTICS – Each actual parameter shall resolve to a specific value compatible with the type of its corresponding formal parameter, or in the case of predefined operations compatible with the types for which the operation is defined. */

/* STATIC SEMANTICS – If a parameter is a parameterized constraint then the constraint, shall be passed together with its actual parameter list. */

/* STATIC SEMANTICS – The actual parameters shall be bound. */

/* STATIC SEMANTICS – If the type of the formal parameter is PDU, then the actual parameter's type shall be declared as PDU or as a specific **PDU** type. */

700 ActualPar ::= Value | PCO_Identifier | CP_Identifier | TimerIdentifier

/* NOTE – Through Value, it is possible to reach MatchingSymbol, TS_ParIdentifier, TS_ConstIdentifier, TS_VarIdentifier, TC_VarIdentifier, FormalParIdentifier or ConsRef. */

A.3.3.34 Expressions

701 AssignmentList ::= "(" Assignment {Comma Assignment} ")"

702 Assignment ::= DataObjectReference ":=" Expression

/* STATIC SEMANTICS – Except within a Procedural Definition or an Encoding Definition, the LHS of Assignment shall only resolve to: TS_VarIdentifier, TC_VarIdentifier, reference to the field of a variable or reference to an ASP parameter or PDU field that is to be sent. */

/* STATIC SEMANTICS – Within a procedure definition of a TSOp or EncodingOp, the DataObject Identifier on the left-hand side of an assignment shall be a VarIdentifier. */

/* STATIC SEMANTICS – The expression shall contain no unbound variables. */

/* OPERATIONAL SEMANTICS – The Expression on the RHS of Assignment shall evaluate to an explicit value of the type of the LHS. */

703 Expression ::= SimpleExpression [RelOp SimpleExpression]

/* OPERATIONAL SEMANTICS – If both SimpleExpressions and the RelOp exist, then the SimpleExpressions shall evaluate to specific values of compatible types. */

/* OPERATIONAL SEMANTICS – If RelOp is "<" | ">" | ">=" | "<=", then each SimpleExpression shall evaluate to a specific INTEGER value. */

/* STATIC SEMANTICS – ASN.1 Named Values shall not be used within arithmetic expressions as operands of operations. */

704 SimpleExpression ::= Term {AddOp Term}

/* OPERATIONAL SEMANTICS – Each Term shall resolve to a specific value. If more than one Term exists and if AddOp is "OR", then the Terms shall resolve to type BOOLEAN; if AddOp is "+" or "–", then the Terms shall resolve to type INTEGER. */

705 Term ::= Factor {MultiplyOp Factor}

/* OPERATIONAL SEMANTICS – Each Factor shall resolve to a specific value. If more than one Factor exists and if MultiplyOp is "AND", then the Factors shall resolve to type BOOLEAN; if MultiplyOp is "*" or "/", then the Factors shall resolve to type INTEGER. */

706 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. */

707 Primary ::= Value | DataObjectReference | OpCall | SelectExprIdentifier | "(" Expression ")"

/* STATIC SEMANTICS - SelectExprIdentifier shall only be used within selection expressions. */

/* NOTE – Through Value, it is possible to reach MatchingSymbol, TS_ParIdentifier, TS_ConstIdentifier, TS_VarIdentifier, TC_VarIdentifier, FormalParIdentifier or ConsRef. */

708 DataObjectReference ::= DataObjectIdentifier {ComponentReference}

/* STATIC SEMANTICS – Identifiers of ASP parameters and PDU fields associated with SEND and RECEIVE shall be used only to reference ASP parameter and PDU field values on the statement line itself. */

/* STATIC SEMANTICS – Each ComponentReference shall only reference an ASP parameter, PDU field, structure element or ASN.1 value explicitly declared in the object that immediately precedes in the DataObjectReference. */

/* STATIC SEMANTICS – DataObjectIdentifier shall not be a VarIdentifier except within a procedure definition of a TestSuiteOperation or EncodingOperation. */

709 DataObjectIdentifier ::= TS_ParIdentifier | TS_ConstIdentifier | TS_VarIdentifier | TC_VarIdentifier | FormalParIdentifier | ASP_Identifier | PDU_Identifier | CM_Identifier | VarIdentifier

710 ComponentReference ::= RecordRef | ArrayRef | BitRef

/* STATIC SEMANTICS – RecordRef shall be used to reference ASN.1 SEQUENCE, SET and CHOICE components. It shall not be used to reference components of any other ASN.1 type. */

/* STATIC SEMANTICS – RecordRef shall be used to reference ASP parameters, PDU fields and structure elements in the tabular form. */

/* STATIC SEMANTICS – ArrayRef shall be used to reference ASN.1 SEQUENCE OF and SET OF components. It shall not be used to reference components of any other ASN.1 type. */

711 RecordRef ::= Dot (ComponentIdentifier | ComponentPosition)

/* STATIC SEMANTICS – The ComponentIdentifier form of RecordRef shall always be used to reference ASN.1 SEQUENCE, SET and CHOICE components when an identifier is declared for the component. */

/* STATIC SEMANTICS – The ComponentIdentifier form of RecordRef shall always be used to reference ASP parameters, PDU fields and structure elements declared in the tabular form. */

/* STATIC SEMANTICS – The ComponentPosition form of RecordRef shall always be used to reference ASN.1 SEQUENCE, SET and CHOICE components when an identifier is not declared for the component. */

/* STATIC SEMANTICS – StructIdentifier shall not be used if the relevant structure is used as a macro. StructIdentifiers and PDU_Identifiers shall not be included in a RecordRef when a parameter, field or element is chained to a PDU or structure and the RecordRef is to identify a component of that PDU or structure. */

/* STATIC SEMANTICS – Where a structure is used as a macro expansion, the elements in the structure shall be referred to as if it was expanded into the ASP or PDU referring to it. */

/* STATIC SEMANTICS – If a parameter, field or element is defined to be of metatype PDU no reference shall be made to fields of that substructure. */

712 ComponentIdentifier ::= ASP_ParIdentifier | PDU_FieldIdentifier | CM_ParIdentifier | ElemIdentifier | ASN1_Identifier

713 ASN1_Identifier ::= Identifier

/* NOTE – ASN1_Identifier identifies a field within ASN.1 SEQUENCE, SET or CHOICE type. */

/* STATIC SEMANTICS – An ASN1_Identifier associated with a NamedValue shall not be used unless the value is within a SEQUENCE, SET or CHOICE type. */

/* STATIC SEMANTICS – An ASN1_Identifier shall be provided to identify the variant in a CHOICE type. */

/* STATIC SEMANTICS – An ASN1_Identifier shall be provided whenever the value definition becomes ambiguous because of omitted OPTIONAL values in a SEQUENCE type. */

714 ComponentPosition ::= "(" Number ")"

715 ArrayRef ::= Dot "[" ComponentNumber "]"

716 ComponentNumber ::= Expression

/* OPERATIONAL SEMANTICS – ComponentNumber shall evaluate to a non-negative specific INTEGER value. */

717 BitRef ::= Dot (BitIdentifier | "[" BitNumber "]")

718 BitIdentifier ::= Identifier

/* NOTE – BitIdentifier identifies a particular bit within an ASN.1 BIT STRING. */

719 BitNumber ::= Expression

/* OPERATIONAL SEMANTICS – BitNumber shall evaluate to a non-negative specific INTEGER value. */

720 OpCall ::= OpIdentifier (ActualParList | "(" ")")

/* STATIC SEMANTICS – See static semantics on production 699. */

721 OpIdentifier ::= TS_OpIdentifier | TS_ProcIdentifier | PredefinedOpIdentifier

722 PredefinedOpIdentifier ::= BIT_TO_INT | HEX_TO_INT | INT_TO_BIT | INT_TO_HEX | IS_CHOSEN | IS_PRESENT | LENGTH_OF | NUMBER_OF_ELEMENTS

723 AddOp ::= "+" | "–" | **OR**

/* OPERATIONAL SEMANTICS – Operands of the "+", "–" operators shall be of type INTEGER (i.e., TTCN or ASN.1 predefined) or derivations of INTEGER (i.e., subrange). Operands of the OR operator shall be of type BOOLEAN (TTCN or ASN.1 predefined) or derivatives of BOOLEAN. */

724 MultiplyOp ::= "*" | "/" | **MOD** | **AND**

/* OPERATIONAL SEMANTICS – Operands of the "*", "/" and MOD operators shall be of type INTEGER (i.e., TTCN or ASN.1 predefined) or derivations of INTEGER (i.e., subrange). Operands of the AND operator shall be of type BOOLEAN (TTCN or ASN.1 predefined) or derivatives of BOOLEAN. */

725 UnaryOp ::= "+" | "–" | **NOT**

/* OPERATIONAL SEMANTICS – Operands of the "+", "–" operators shall be of type INTEGER (i.e., TTCN or ASN.1 predefined) or derivations of INTEGER (i.e., subrange). Operands of the NOT operator shall be of type BOOLEAN (TTCN or ASN.1 predefined) or derivatives of BOOLEAN. */

726 RelOp ::= "=" | "<" | ">" | "<>" | "≥" | "≤"

A.3.3.35 Timer operations

727 TimerOps ::= TimerOp {Comma TimerOp}

728 TimerOp ::= StartTimer | CancelTimer | ReadTimer

729 StartTimer ::= **START** (TimerIdentifier | FormalParIdentifier) ["(" TimerValue ")"]

/* STATIC SEMANTICS – FormalParIdentifier shall only be of TIMER type. */

730 CancelTimer ::= **CANCEL** [TimerIdentifier | FormalParIdentifier]

/* STATIC SEMANTICS – FormalParIdentifier shall only be of TIMER type. */

731 TimerValue ::= Expression

/* OPERATIONAL SEMANTICS – Timervalue shall evaluate to a non-zero positive INTEGER. */

732 ReadTimer ::= **READTIMER** (TimerIdentifier | FormalParIdentifier) "(" DataObjectReference ")"

/* STATIC SEMANTICS – FormalParIdentifier shall only be of TIMER type. */

/* STATIC SEMANTICS – The DataObjectReference shall only resolve to TS_VarIdentifier, TC_VarIdentifier, or reference to the field of a variable. */

/* OPERATIONAL SEMANTICS – The DataObjectReference shall resolve to type INTEGER. */

A.3.3.36 Types

733 TypeOrPDU ::= Type | **PDU**
734 Type ::= PredefinedType | ReferenceType

A.3.3.36.1 Predefined types

735 PredefinedType ::= **INTEGER** | **BOOLEAN** | **BITSTRING** | **HEXSTRING** | **OCTETSTRING** | **OBJECTIDENTIFIER** | **R_Type** | CharacterString
736 CharacterString ::= **NumericString** | **PrintableString** | **TeletexString** | **VideotexString** | **VisibleString** | **IA5String** | **GraphicString** | **GeneralString** | **T61String** | **ISO646String**

A.3.3.36.2 Referenced types

737 ReferenceType ::= TS_TypeIdentifier | ASP_Identifier | PDU_Identifier | CM_Identifier
/* STATIC SEMANTICS – All types, other than the predefined types, used in a test suite shall be declared in the Test Suite Type definitions, ASP type definitions, PDU type definitions or CM type definitions, and referenced by name. */
738 TS_TypeIdentifier ::= SimpleTypeIdentifier | StructIdentifier | ASN1_TypeIdentifier

A.3.3.37 Values

739 Value ::= LiteralValue | ASN1_Value [ASN1_Encoding]
/* REFERENCE – Where ASN1_Value is the non-terminal Value as defined in Recommendation X.680. For the purposes of TTCN, the following production defined in Recommendation X.680:
DefinedValue ::= Externalvaluereference | valuereference is redefined to be:
DefinedValue ::= ConstraintValue&Attributes | valuereference
This means that ASN.1 external references are not allowed in TTCN, but the full possibilities of ConstraintValue&Attributes as defined in production 562 are allowed within ASN.1 values in TTCN. This means that expressions, matching symbols, constraint references, value lengths, IF_PRESENT, and ASN.1 field encoding operations are all included . */

For the purpose of TTCN, the following productions in Recommendation X.680:
BuiltinValue ::=
BitStringValue |
BooleanValue |
CharacterStringValue |
ChoiceValue |
EmbeddedPDUValue |
EnumeratedValue |
ExternalValue |
InstanceOfValue |
IntegerValue |
NullValue |
ObjectClassFieldValue |
ObjectIdentifierValue |
OctetStringValue |
RealValue |
SequenceOfValue |
SequenceOfValue |
SetValue |
SetOfValue |
TaggedValue

ReferencedValue ::=
DefinedValue |
ValueFromObject

are redefined to be:

BuiltinValue ::=
BitStringValue |
BooleanValue |
CharacterStringValue |
ChoiceValue |
EmbeddedPDUValue |
EnumeratedValue |
ExternalValue |
IntegerValue |
NullValueValue |
ObjectIdentifiervalue |

OctetStringValue |
RealValue |
SequenceValue |
SequenceOfValue |
SetValue |
SetOfValue |
TaggedValue

ReferencedValue ::=
DefinedValue */

/* STATIC SEMANTICS – ASN.1 Named Values shall not be used within arithmetic expressions as operands of operations. */

740 LiteralValue ::= Number | BooleanValue | Bstring | Hstring | Ostring | Cstring | R_Value

741 Number ::= (NonZeroNum {Num}) | **0**

742 NonZeroNum ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

743 Num ::= **0** | NonZeroNum

744 BooleanValue ::= **TRUE** | **FALSE**

745 Bstring ::= "_" {Bin | Wildcard} "_" **B**

746 Bin ::= **0** | **1**

747 Hstring ::= "_" {Hex | Wildcard} "_" **H**

748 Hex ::= Num | **A** | **B** | **C** | **D** | **E** | **F**

749 Ostring ::= "_" {Oct | Wildcard} "_" **O**

750 Oct ::= Hex Hex

751 Cstring ::= """ {Char | Wildcard | "\"} """

752 Char ::= /* *REFERENCE – A character defined by the relevant CharacterString type*. */

/* LEXICAL REQUIREMENT – If the CharacterString type includes the character " (double quote), this character shall be represented by a pair of " (double quote) in the denotation of any value. */

753 Wildcard ::= AnyOne | AnyOrNone

754 AnyOne ::= "?"

/* STATIC SEMANTICS – AnyOne shall be used only within values of string types, SEQUENCE OF and SET OF. */

755 AnyOrNone ::= "*"

/* STATIC SEMANTICS – AnyOrNone shall be used only within values of string types, SEQUENCE OF and SET OF. */

756 R_Value ::= **pass** | **fail** | **inconc** | **none**

757 Identifier ::= Alpha{AlphaNum | Underscore | DoubleColon}

/* STATIC SEMANTICS – All Identifiers referenced in a TTCN test suite shall be explicitly declared in the test suite, explicitly declared in an ASN.1 type definition referenced by the test suite or be a TTCN predefined identifier. */

/* STATIC SEMANTICS – DoubleColon shall only be used in identifiers which are declared in an Import table. Identifiers containing DoubleColon shall not appear in an Export table. The DoubleColon is used to separate the name of a TTCN Module from an identifier originally specified in that TTCN Module. */

758 Alpha ::= UpperAlpha | LowerAlpha

759 AlphaNum ::= Alpha | Num

760 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**

761 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**

762 ExtendedAlphaNum ::= */* *REFERENCE – A character from any character set defined in ISO/IEC 10646.* */*

763 BoundedFreeText ::= *"/*"* FreeText *"*/"*

764 FreeText ::= {ExtendedAlphaNum}

/* LEXICAL REQUIREMENT – Free Text shall not contain the string "*/" unless preceded by backslash ("\"). */

A.3.3.38 Miscellaneous productions

765 Comma ::= ","

766 Dot ::= "."

767 Dash ::= "-"

768 Minus ::= "-"

769 SemiColon ::= ";"

770 DoubleColon ::= Colon Colon

771 Colon ::= ":"

772 Underscore ::= "_"

A.4 General static semantics requirements

A.4.1 Introduction

Static semantics requirements that are related to specific BNF productions are specified as comments on the relevant productions, in the following format:

/* STATIC SEMANTICS – ... ***/**

All other static semantic requirements that are common to both TTCN.GR and TTCN.MP are specified in the remainder of A.4. Additional static semantics in the TTCN.MP are specified in A.5.2.

A.4.2 Uniqueness of identifiers

A.4.2.1 In some cases test suites may make references to items defined in other OSI Recommendations. In particular, references to ASN.1 type definition modules according to Recommendation X.680 may be made in the type definitions. Names from those modules (such as identifiers of subfields within structured ASN.1 type definitions) may be used throughout the test suite.

Since the rules for identifiers in ASN.1 and TTCN conflict, the following conventions apply:

a) type references, module identifiers and value references made within the various ASN.1 type definitions tables shall comply to the requirements for identifiers defined in Recommendation X.680;

b) for identifiers used within the other parts of a test suite dash (-) characters shall be replaced with underscores (_).

Within some TTCN tables part of the ASN.1syntax can be used to define types.In that case, ASN.1 rules shall be followed for identifiers, with the exception that dash (-) characters shall not be used. Underscores (_) may be used instead. All other requirements defined by Recommendation X.680 (e.g., Type identifiers shall start with an upper-case letter, and field identifiers within structured ASN.1 definitions shall start with a lower-case letter) apply to TTCN test suites wherever ASN.1 is used.

A.4.2.2 All identifiers of the following TTCN objects shall be unique throughout the test suite:

a) Test Suite Types.

b) Test Suite Operations.

c) Test Suite Parameters.

d) Test Case Selection Expressions.

e) Test Suite Constants.

f) Test Suite Variables.

g) Test Case Variables.

h) PCO types.

NOTE – If there is no PCO type declaration table, then PCO types are implicitly declared in the PCO declaration table, in which case the uniqueness refers to the meaning of the PCO type – the same PCO type may occur several times in the PCO declaration table with the same meaning.

i) PCOs.

j) CPs.

k) Timers.

l) Test Components.

m) Test Component Configurations.

n) ASP types.

o) PDU types.

p) CM types.

q) Structured Types.

r) Encoding Rules.

s) Encoding Variations.

t) Invalid Field Encodings.

u) Aliases.

v) ASP constraints.

w) PDU constraints.

x) CM constraints.

y) Structure constraints.

z) Test Cases.

aa) Test Steps.

ab) Defaults.

ac) Encoding Rule Names.

ad) Encoding Variation Names.

ae) Invalid Field Encoding Names.

A.4.2.3 All the following TTCN object references shall be unique throughout the test suite:

a) Test Group References.

b) Test Step Group References.

c) Default Group References.

A.4.2.4 TTCN reserved words are listed in Table A.2. These reserved words shall not be used as identifiers in a TTCN test suite. All TTCN reserved words and TTCN identifiers are case sensitive.

A.4.2.5 The ASN.1 reserved words are listed in Table A.3. These reserved words shall not be used as identifiers in a TTCN test suite.

A.4.2.6 When ASN.1 is used in a TTCN test suite, ASN.1 identifiers from the following list shall be unique throughout the test suite, regardless of whether the ASN.1 definition is explicit or implicit by reference:

a) *TypeIdentifiers* of an ASN.1 Type Definition;

b) identifiers occurring in an ASN.1 ENUMERATED type as distinguished values;

c) identifiers occurring in a *NamedNumberList* of an ASN.1 INTEGER type.

A.4.2.7 The names of ASP parameters shall be unique within the ASP in which they are declared. The names of PDU fields shall be unique within the PDU in which they are declared. The names of CM parameters shall be unique within the CM in which they are declared.

A.4.2.8 If a Structured Type is used as a macro expansion, then the names of the elements within the Structured Type shall be unique within each ASP, PDU or CM where it will be expanded.

A.4.2.9 Labels used within a tree shall be unique within a tree (i.e., Test Case root tree, Test Step tree, Default tree, local tree).

A.4.2.10 The tree header identifier used for local trees shall be unique within the dynamic behaviour description in which they appear, and shall not be the same as any identifier having a unique meaning throughout the test suite.

NOTE – This means that a local tree identifier may have the same name as a local tree identifier in another behaviour description, but not the same as another Test Step in the Test Step Library.

A.4.2.11 The formal parameter names which may optionally appear as part of the following shall be unique within that formal parameter list, and shall not be the same as any identifier having a unique meaning throughout the test suite:

a) Test suite operations definition;

b) Tree header of a local tree;

c) Test Step Identifier;

d) Default Identifier;

e) Parameterized constraint declaration.

Table A.2/X.292 – TTCN Reserved Words

ACTIVATE	IA5String	pass
AND	IF	PDU
BEGIN	IF_PRESENT	PERMUTATION
BITSTRING	INCONC	PrintableString
BIT_TO_INT	inconc	ps
BOOLEAN	INFINITY	PTC
BY	INTEGER	R
CANCEL	INT_TO_BIT	READTIMER
CASE	INT_TO_HEX	REPEAT
COMPLEMENT	IS_CHOSEN	REPLACE
CP	IS_PRESENT	RETURN
CREATE	IUT	RETURNVALUE
DO	LT	R_Type
DONE	min	s
ELSE	MOD	START
ENC	ms	STATIC
END	MTC	SUPERSET
ENDCASE	NOT	SUBSET
ENDIF	ns	TeletexString
ENDVAR	OF	THEN
ENDWHILE	OMIT	TIMEOUT
F	OR	TIMER
FAIL	OTHERWISE	TO
fail	P	TRUE
FALSE	LENGTH_OF	UNTIL
GeneralString	none	μs
GOTO	NUMBER_OF_ELEMENTS	UT
GraphicString	NumericString	VAR
HEXSTRING	OCTETSTRING	VideotexString
HEX_TO_INT	OBJECTIDENTIFIER	VisibleString
I	PASS	WHILE

A.4.2.12 A formal parameter name contained in the formal parameter list of a local tree header shall take precedence over a formal parameter name contained in the formal parameter list of the Test Step in which it is defined, within the scope of that local formal parameter list.

A.4.2.13 In concurrent TTCN, PCOs and CPs used in a Test Case shall only be those determined by the Test Component configuration for that Test Case.

A.4.2.14 Each identifier used in the procedural definition of a Test Suite Operation shall be on of the following:

a) locally declared variable name;

b) a type name, used in a variable declaration;

c) a formal parameter name declared in a formal parameter list of the operation;

d) a Test Suite Operation name.

Table A.3/X.292 – ASN.1 Reserved Words

ABSENT	EXTERNAL	OPTIONAL
ABSTRACT SYNTAX	FALSE	PDV
ALL	FROM	PRESENT
APPLICATION	GeneralString	PRIVATE
AUTOMATIC	GeneralizedTime	PrintableString
BEGIN	GraphicString	REAL
BIT	IA5String	SEQUENCE
BMPString	IDENTIFIER	SET
BOOLEAN	IMPLICIT	SIZE
CHARACTER	IMPORT	STRING
CHOICE	INCLUDES	SYNTAX
CLASS	INSTANCE	T61String
COMPONENT	INTEGER	TRUE
COMPONENTS	INTERSECTION	TeletexString
CONSTRAINED	ISO646String	TYPEIDENTIFIER
DEFAULT	MAX	UNION
DEFINITIONS	MIN	UNIQUE
EMBEDDED	NULL	UNIVERSAL
END	NumericString	Universalstring
ENUMERATED	OBJECT	UTCTime
EXCEPT	ObjectDescriptor	VideotexString
EXPLICIT	OCTET	VisibleString
EXPORT	OF	WITH

NOTE – Table A.3 contains a number of keywords which at present have no support within this Recommendation. Those keywords have been reserved to facilitate future integration of ASN.1 1997 features into TTCN.

The scope of formal parameter names and locally declared variable names is the procedural definition of the Test Suite Operation. Thus, the values of all other types of identifier are not directly accessible within the procedural definition of a test suite operation. To access such values they shall be passed as actual parameters to the Test Suite Operation.

A.4.2.15 The constraints for TTCN Structured Types, TTCN ASPs, TTCN PDUs and TTCN CMs shall not be specified using ASN.1 tables (i.e. ASN.1 Type Constraints, ASN.1 ASP Constraints, ASN.1 PDU Constraints or ASN.1 CM Constraints). Conversely, the constraints for ASN.1 Types, ASN.1 ASPs, ASN.1 PDUs and ASN.1 CMs shall not be specified using TTCN tables (i.e. Structured Type Constraints, TTCN ASP Constraints, TTCN PDU Constraints or TTCN CM Constraints).

NOTE – However, when ASPs or PDUs are chained to other PDUs, the enclosing ASP or PDU may, for example, be specifiied in tabular TTCN, whereas the enclosed PDU may be specified in ASN.1.

A.5 Differences between TTCN.GR and TTCN.MP

A.5.1 Differences in syntax

The following is a list of syntax differences between TTCN.MP and TTCN.GR:

a) TTCN.MP uses keywords as delimiters between entries, while TTCN.GR uses boxes;

b) TTCN.MP uses an explicit denotation of indentation levels for test events, while indentation is indicated visually in TTCN.GR;

c) TTCN.MP contains an extra occurrence of the suite identifier, which is used to facilitate identification of the ATS in an automated method;

d) in TTCN.MP the Test Case behaviour descriptions are explicitly grouped by the inclusion of appropriate Test Group Identifiers in sequence before the Test Case behaviour descriptions belonging to each group; this information duplicates information contained in the Test Case Index and in the Test Group References of the Test Case behaviour descriptions;

e) the Test Suite Structure, Test Case Index, Test Step Index and Default Index tables require a page number for each entry; since page numbers are not relevant in the machine processable form, they are not reflected in the TTCN.MP;

f) TTCN.GR supports both single and compact proformas for ASP and PDU constraints and Test Cases; the TTCN only supports BNF for the single table format and the presentation of a number of single tables in TTCN.GR compact format is a display issue; when mapping a compact constraints table to TTCN.MP (i.e., single format), blank fields due to modification shall be omitted;

g) the symbols "/*" and "*/" which open and close BoundedFreeText strings in the TTCN.MP shall not appear in the TTCN.GR;

h) there are two alternative positions for the labels column in behaviour description tables in TTCN.GR, whereas there is a fixed position for the labels in TTCN.MP;

i) page and line continuation are TTCN.GR features which are not represented in the TTCN.MP;

j) page and line numbering are TTCN.GR features which are not represented in the TTCN.MP;

k) if in TTCN.GR group references are used with definitions, declarations or constraints to indicate an hierarchical grouping of objects, then in TTCN.MP each relevant group identifier is inserted before the syntax for the group of tables which share that group identifier and the syntax for the group identifier and following group of tables are enclosed in the appropriate TTCN.MP keywords, relevant to the type of object.

A.5.2 Additional static semantics in the TTCN.MP

The following is a list of the additional static semantics in the TTCN.MP:

a) in the TTCN.MP, statements in the first level of alternatives having no predecessor in the root or local tree they belong have the indentation value of zero; statements having a predecessor shall have the indentation value of the predecessor plus one as their indentation value;

b) in the TTCN.MP, the Test Suite Structure information in the form of Test Group Identifiers preceding Test Case behaviour descriptions shall be the same structure as defined by the part of the Test Suite Structure relevant to Test Groups and that defined by the Test Case Index.

A.6 List of BNF production numbers

A.6.1 Introduction

This subclause presents an alphabetical index of the BNF productions that appear in Annex A. For each production the index gives a reference in terms of the production number (not page number).

A.6.2 The production index

A

Activate 690

ActualCrefPar 672

ActualCrefParList 671

ActualPar 700

ActualParList 699

AddOp 723

AliasDef 495

AliasDefs 491

AliasDefsGroup 488

AliasDefsGroupId 489

AliasDefsGroupIdentifier 490

AliasDefsOrGroup 487

AliasGroupIdentifier 494

AliasGroupRef 492

AliasGroupReference 493

AliasId 496

AliasIdentifier 497

Alpha 758

AlphaNum 759

AnyOne 754

AnyOrNone 755

AnyOrOmit 569

AnyValue 568

ArrayRef 715

ASN_ASP_DefsByRefGroupRef 373

ASN1_ASP_Constraint 540

ASN1_ASP_ConstraintGroup 538

ASN1_ASP_ConstraintGroupId 539

ASN1_ASP_ConstraintGroupIdentifier 543

ASN1_ASP_ConstraintGroupRef 541

ASN1_ASP_ConstraintGroupReference 542

ASN1_ASP_ConstraintOrGroup 537

ASN1_ASP_Constraints 536

ASN1_ASP_GroupIdentifier 368

ASN1_ASP_GroupRef 366

ASN1_ASP_GroupReference 367

ASN1_ASP_TypeDef 365

ASN1_ASP_TypeDefByRef 374

ASN1_ASP_TypeDefGroup 363

ASN1_ASP_TypeDefGroupId 364

ASN1_ASP_TypeOrGroup 362

ASN1_ASP_TypeDefs 361

ASN1_ASP_TypeDefsByRef 372

ASN1_ASP_TypeDefsByRefGroup 370

ASN1_ASP_TypeDefsByRefGroupId 371

ASN1_ASP_TypeDefsByRefOrGroup 369

ASN1_CM_Constraint 613

ASN1_CM_ConstraintGroup 611

ASN1_CM_ConstraintGroupId 612

ASN1_CM_ConstraintGroupIdentifier 616

ASN1_CM_ConstraintGroupRef 614

ASN1_CM_ConstraintGroupReference 615

ASN1_CM_ConstraintOrGroup 610

ASN1_CM_Constraints 609

ASN1_CM_GroupIdentifier 441

ASN1_CM_GroupRef 439

ASN1_CM_GroupReference 440

ASN1_CM_TypeDef 438

ASN1_CM_TypeDefGroup 436

ASN1_CM_TypeDefGroupId 437

ASN1_CM_TypeDefOrGroup 435

ASN1_CM_TypeDefs 434

ASN1_ConsValue 594

ASN1_Encoding 579

ASN1_Identifier 713

ASN1_LocalType 123

ASN1_ModeleId 132

ASN1_ModeleIdentifier 133

ASN1_PDU_Constraint 590

ASN1_PDU_ConstraintGroup 588

ASN1_PDU_ConstraintGroupId 589

ASN1_PDU_ConstraintGroupIdentifier 593

ASN1_PDU_ConstraintGroupRef 591

ASN1_PDU_ConstraintGroupReference 592

ASN1_PDU_ConstraintOrGroup 587

ASN1_PDU_Constraints 586

ASN1_PDU_DefsByRefGroupRef 415

ASN1_PDU_GroupIdentifier 410

ASN1_PDU_GroupRef 408

ASN1_PDU_GroupReference 409

ASN1_PDU_TypeDef 407

ASN1_PDU_TypeDefByRef 416
ASN1_PDU_TypeDefs 403
ASN1_PDU_TypeDefsByRef 414
ASN1_PDU_TypeDefsByRefGroup 412
ASN1_PDU_TypeDefsByRefGroupId 413
ASN1_PDU_TypeDefsByRefOrGroup 411
ASN1_Type 122
ASN1_Type&LocalTypes 121
ASN1_TypeConstraint 521
ASN1_TypeConstraintGroup 519
ASN1_TypeConstraintGroupId 520
ASN1_TypeConstraintGroupIdentifier 524
ASN1_TypeConstraintGroupRef 522
ASN1_TypeConstraintGroupReference 523
ASN1_TypeConstraintOrGroup 518
ASN1_TypeConstraints 517
ASN1_TypeDef 113
ASN1_TypeDefinition 120
ASN1_TypeDefOrGroup 110
ASN1_TypeDefs 109
ASN1_TypeGroup 111
ASN1_TypeGroupId 112
ASN1_TypeGroupIdentifier 119
ASN1_TypeGroupRef 117
ASN1_TypeGroupReference 118
ASN1_TypeId 114
ASN1_TypeId&FullId 115
ASN1_TypeIdentifier 116
ASN1_TypeRef 129
ASN1_TypeReference 130
ASN1_TypeRefs 127
ASN1_TypeRefsGroup 125
ASN1_TypeRefsGroupId 126
ASN1_TypeRefsGroupRef 128
ASN1_TypeRefOrGroup 124
ASN1_ValueReference 227
ASP_ConstraintGroupIdentifier 533
ASP_ConstraintGroupRef 531
ASP_ConstraintGroupReference 532
ASP_Constraints 525
ASP_GroupIdentifier 352
ASP_GroupRef 350
ASP_GroupReference 351
ASP_Id 347
ASP_Id&FullId 348
ASP_Identifier 349
ASP_ParDcl 355
ASP_ParDcls 354
ASP_ParId 356
ASP_ParId&FullId 358
ASP_ParIdentifier 359
ASP_ParIdOrMacro 357
ASP_ParType 360
ASP_ParValue 535
ASP_ParValues 534
ASP_TypeDefs 341
Assignment 702
AssignmentList 701
Attach 696

B

BehaviourDescription 653
BehaviourLine 663
Bin 746
BitIdentifier 718
BitNumber 719
BitRef 717
BooleanValue 744
Bound 399
BoundedFreeText 763
Bstring 745

C

C_Role 316
CancelTimer 730
CaseClause 173
CaseIndex 59
CaseStatement 172
Char 752
CharacterString 736
CM_ConstraintGroupIdentifier 606
CM_ConstraintGroupRef 604

CM_ConstraintGroupReference 605
CM_Constraints 598
CM_GroupIdentifier 427
CM_GroupRef 425
CM_GroupReference 426
CM_Id 423
CM_Identifier 424
CM_ParDcl 429
CM_ParDcls 428
CM_ParId 430
CM_ParIdentifier 432
CM_ParIdOrMacro 431
CM_ParType 433
CM_ParValue 608
CM_ParValues 607
CM_TypeDefs 417
CollComment 58
Colon 771
Comma 765
Comment 53
Complement 566
ComplexDefinitions 340
ComponentIdentifier 712
ComponentNumber 716
ComponentPosition 714
ComponentReference 710
Configuration 628
ConsId 554
ConsId&ParList 555
ConsRef 670
ConstraintExpression 564
ConstraintIdentifier 556
ConstraintReference 669
ConstraintsPart 500
ConstraintValue 563
ConstraintValue&Attributes 562
ConstraintValue&AttributesOrReplace 595
Construct 689
ConsValue 561
CP_Dcl 288
CP_Dcls 284
CP_DclsGroup 281
CP_DclsGroupId 282
CP_DclsGroupIdentifier 283
CP_DclsOrGroup 280
CP_GroupIdentifier 287
CP_GroupRef 285
CP_GroupReference 286
CP_Id 289
CP_Identifier 290
CP_List 339
CPs_Used 338
Create 692
CreateList 693
CreateTComp 694
Cref 668
Cstring 751

D

Dash 767
DataObjectIdentifier 709
DataObjectReference 708
Declarations 205
DeclarationsPart 67
DeclarationValue 219
Default 646
DefaultExpression 456
DefaultGroup 644
DefaultGroupId 645
DefaultGroupIdentifier 652
DefaultGroupReference 651
DefaultId 648
DefaultId&ParList 649
DefaultIdentifier 650
DefaultIndex 63
DefaultRef 647
DefaultReference 631
DefaultRefList 630
DefaultsLibrary 643
DefaultRef 629
DefaultValue 190
DefIndex 64

Definitions 68
DerivationPath 558
DerivPath 557
Description 60
Done 687
Dot 766
DoubleColon 770
Duration 302
DynamicPart 617

E

ElemDcl 104
ElemDcls 103
ElemId 105
ElemId&FullId 106
ElemIdentifier 107
ElemType 108
ElemValue 513
ElemValues 512
Encoding_TypeList 466
EncodingDefault 455
EncodingDefinition 450
EncodingDefinitions 446
EncodingDefinitionGroup 444
EncodingDefinitionGroupId 445
EncodingDefinitionOrGroup 443
EncodingDefs 442
EncodingGroupIdentifier 449
EncodingGroupRef 447
EncodingGroupReference 448
EncodingRef 453
EncodingReference 454
EncodingRuleId 451
EncodingRuleIdentifier 452
EncodingVariation 468
EncodingVariationId 469
EncodingVariationList 465
EncodingVariations 457
EncodingVariationsSet 461
EncodingVariationSetGroup 459
EncodingVariationSetGroupId 460
EncodingVariationSetOrGroup 458
EncRuleId 553
EncVariationCall 511
EncVariationGroupIdentifier 464
EncVariationGroupRef 462
EncVariationGroupReference 463
EncVariationId 510
EncVariationId&ParList 470
EncVariationIdentifier 471
Event 680
ExpandedId 498
Expansion 499
ExportedObject 10
ExportedObjects 9
Expression 703
ExtendedAlphaNum 762
ExternalGroupId 22
ExternalGroupIdentifier 23
ExternalObject 24
ExternalObjectId 25
ExternalObjectIdentifier 26
ExternalObjects 21

F

Factor 706
Fail 676
FormalPar&Type 660
FormalParIdentifier 661
FormalParList 659
FormalParType 662
FreeText 764
FullIdentifier 98

G

GoTo 695

H

Header 656
Hex 748
Hstring 747

I

Identifier 757
IfStatement 170
ImplicitSend 683
ImportDeclarations 27
ImportedObject 38
ImportedObjects 37
ImportPart 66
Imports 31
ImportsGroup 29
ImportsGroupId 30
ImportGroupIdentifier 35
ImportGroupRef 33
ImportGroupReference 34
ImportOrGroup 28
Inconclusive 677
Indentation 665
IntegerLabel 174
IntegerRange 86
InvalidFieldEncodingCall 516
InvalidFieldEncodingDef 479
InvalidFieldEncodingDefinition 486
InvalidFieldEncodingDefOrGroup 476
InvalidFieldEncodingDefs 475
InvalidFieldEncodingGroup 477
InvalidFieldEncodingGroupId 478
InvalidFieldEncodingGroupIdentifier 485
InvalidFieldEncodingGroupRef 483
InvalidFieldEncodingGroupReference 484
InvalidFieldEncodingId 480
InvalidFieldEncodingId&ParList 481
InvalidFieldEncodingIdentifier 482

L

Label 667
LabelId 666
LengthAttribute 397
LengthRestriction 83
Line 664
LengthRestriction 740
LocalTree 655
LowerAlpha 761
LowerBound 401
LowerRangeBound 573
LowerTypeBound 87
LowerValueBound 584

M

MacroSymbol 392
MatchingSymbol 565
Minus 768
MultiplyOp 724
MuxValue 277

N

NonZeroNum 742
Num 743
Num_CPs 321
Num_PCOs 319
Number 741
NumOf_CPs 320
NumOf_PCOs 318

O

ObjectDirective 18
ObjectId 11
ObjectIdentifier 12
Objective 642
ObjectType 14
ObjectTypeReference 13
Oct 750
Omit 567
OpCall 720
OpIdentifier 721
Ostring 749
Otherwise 685

P

P_Role 278
Parameterization&Selection 176
Pass 675
PCO_Dcl 273

PCO_Dcls 269
PCO_DclsGroup 266
PCO_DclsGroupId 267
PCO_DclsGroupIdentifier 268
PCO_DclsOrGroup 265
PCO_GroupIdentifier 272
PCO_GroupRef 270
PCO_GroupReference 271
PCO_Id 274
PCO_Identifier 275
PCO_List 337
PCO_Role 279
PCO_Type 353
PCO_TypeDcl 261
PCO_TypeDcls 259
PCO_TypeDclsGroup 256
PCO_TypeDclsGroupId 257
PCO_TypeDclsGroupIdentifier 258
PCO_TypeDclsOrGroup 255
PCO_TypeGroupRef 260
PCO_TypeId 262
PCO_TypeId&MuxValue 276
PCO_TypeIdentifier 263
PCOs_Used 336
PDU_ConstraintGroupIdentifier 552
PDU_ConstraintGroupRef 550
PDU_ConstraintGroupReference 551
PDU_Constraints 544
PDU_EncodingId 387
PDU_FieldDcl 389
PDU_FieldDcls 388
PDU_FieldEncoding 514
PDU_FieldEncodingCall 515
PDU_FieldId 390
PDU_FieldId&FullId 393
PDU_FieldIdentifier 394
PDU_FieldIdOrMacro 391
PDU_FieldType 395
PDU_FieldValue 560
PDU_FieldValues 559
PDU_GroupIdentifier 386
PDU_GroupRef 384
PDU_GroupReference 385
PDU_Id 381
PDU_Id&FullId 382
PDU_Identifier 383
PDU_TypeDefs 375
Permutation 577
PICS_PIXITref 191
PICSref 50
PIXITref 51
PredefinedOpIdentifier 722
PredefinedType 735
Primary 707
ProcBlock 175
ProcStatement 168

Q

Qualifier 681

R

R_Value 756
RangeLength 400
RangeTypeLength 85
RangeValueLength 583
ReadTimer 732
Receive 684
RecordRef 711
ReferenceList 597
ReferenceType 737
RelOp 726
Repeat 697
Replacement 596
Restriction 82
Result 678
Return 691
ReturnValueStatement 169
RoleOrComment 264
RootTree 654

S

SelectExpr 203
SelectExprDef 200
SelectExprDefs 196
SelectExprDefsGroup 193
SelectExprDefsGroupId 194
SelectExprDefsGroupIdentifier 195
SelectExprDefsOrGroup 192
SelectExprGroupIdentifier 199
SelectExprGroupRef 197
SelectExprGroupReference 198
SelectExprId 201
SelectExprIdentifier 202
SelectionExpression 204
SelExprId 56
SemiColon 769
Send 682
SimpleExpression 704
SimpleTypeDef 77
SimpleTypeDefinition 80
SimpleTypeDefs 73
SimpleTypeDefsOrGroup 70
SimpleTypeGroup 71
SimpleTypeGroupId 72
SimpleTypeGroupIdentifier 76
SimpleTypeGroupRef 74
SimpleTypeGroupReference 75
SimpleTypeId 78
SimpleTypeIdentifier 79
SimpleValueList 90
SingleLength 398
SingleTypeLength 84
SingleValueLength 581
SourceId 32
SourceIdentifier 17
SourceInfo 16
SourceRef 36
StandardsRef 49
StartTimer 729
StatementLine 67
StepIndex 962
StructId 96
StructId&FullId 97
StructIdentifier 99
StructTypeConstraint 506
StructTypeConstraintGroup 504
StructTypeConstraintGroupId 505
StructTypeConstraintGroupIdentifier 509
StructTypeConstraintGroupRef 507
StructTypeConstraintGroupReference 508
StructTypeConstraintOrGroup 503
StructTypeConstraints 50
StructTypeDef 295
StructTypeDefOrGroup 92
StructTypeDefs 91
StructTypeGroup 93
StructTypeGroupId 94
StructTypeGroupIdentifier 102
StructTypeGroupRef 100
StructTypeGroupReference 101
Structure&Objective 55
Structure&Objectives 54
SubSet 576
Suite 39
SuiteId 40
SuiteIdentifier 41
SuiteOverviewPart 42
SuiteStructure 48
SuperSet 575

T

TC_VarDcl 250
TC_VarDcls 246
TC_VarDclsGroup 243
TC_VarDclsGroupId 244
TC_VarDclsGroupIdentifier 245
TC_VarDclsOrGroup 242
TC_VarGroupIdentifier 249
TC_VarGroupRef 247
TC_VarGroupReference 248
TC_VarId 251

TC_VarIdentifier 252
TC_VarType 253
TC_VarValue 254
TCompConfigDcl 327
TCompConfigDclGroup 324
TCompConfigDclGroupId 325
TCompConfigDclGroupIdentifier 326
TCompConfigDclOrGroup 323
TCompConfigDcls 322
TCompConfigGroupIdentifier 332
TCompConfigGroupRef 330
TCompConfigGroupReference 331
TCompConfigId 328
TCompConfigIdentifier 329
TCompConfigInfo 334
TCompConfigInfos 333
TCompDcl 313
TCompDcls 309
TCompDclsGroup 306
TCompDclsGroupId 307
TCompDclsGroupIdentifier 308
TcompDclsOrGroup 305
TcompGroupIdentifier 312
TcompGroupRef 310
TcompGroupReference 311
TcompId 314
TcompIdentifier 315
TcompIdList 688
TcompRole 317
TcompUsed 335
Term 705
TestCase 622
TestCaseId 623
TestCaseIdentifier 624
TestCaseIndex 57
TestCases 618
TestGroup 619
TestGroupId 620
TestGroupIdentifier 621
TestGroupRef 625
TestGroupReference 626
TestMethodes 52
TestPurpose 627
TestStep 636
TestStepGroup 633
TestStepGroupId 634
TestStepGroupIdentifier 635
TestStepGroupReference 641
TestStepId 637
TestStepId&ParList 638
TestStepIdentifier 639
TestStepIndex 61
TestStepLibrary 632
TestStepRef 640
TestSuiteExports 65
Timeout 686
TimerDcl 299
TimerDcls 295
TimerDclsGroup 292
TimerDclsGroupId 293
TimerDclsGroupIdentifier 294
TimerDclsOrGroup 291
TimerGroupIdentifier 298
TimerGroupRef 296
TimerGroupReference 297
TimerId 300
TimerIdentifier 301
TimerOp 728
TimerOps 727
TimerValue 731
TimeUnit 304
To 89
TreeHeader 657
TreeIdentifier 658
TreeReference 698
TS_ConstDcl 214
TS_ConstDcls 210
TS_ConstDclsGroup 207
TS_ConstDclsGroupId 208
TS_ConstDclsGroupIdentifier 209

TS_ConstDclsOrGroup 206
TS_ConstGroupIdentifier 213
TS_ConstGroupRef 211
TS_ConstGroupReference 212
TS_ConstId 215
TS_ConstIdentifier 216
TS_ConstRef 226
TS_ConstRefs 224
TS_ConstRefsGroup 221
TS_ConstRefsGroupId 222
TS_ConstRefsGroupIdentifier 223
TS_ConstRefsGroupRef 225
TS_ConstRefsOrGroup 220
TS_ConstType 217
TS_ConstValue 218
TS_OpDef 139
TS_OpDefGroup 136
TS_OpDefGroupId 137
TS_OpDefGroupIdentifier 138
TS_OpDefOrGroup 135
TS_OpDefs 134
TS_OpDescription 147
TS_OpGroupIdentifier 145
TS_OpGroupReference 144
TS_OpId 140
TS_OpId&ParList 141
TS_OpIdentifier 142
TS_OpProcDef 162
TS_OpResult 146
TS_ParDcl 185
TS_ParDcls 181
TS_ParDclsGroup 178
TS_ParDclsGroupId 179
TS_ParDclsGroupIdentifier 180
TS_ParDclsOrGroup 177
TS_ParDefault 189
TS_ParGroupIdentifier 184
TS_ParGroupRef 182
TS_ParGroupReference 183
TS_ParId 186
TS_ParIdentifier 187
TS_ParType 188
TS_ProcDef 153
TS_ProcDefGroup 150
TS_ProcDefGroupId 151
TS_ProcDefGroupIdentifier 152
TS_ProcDefOrGroup 149
TS_ProcDefs 148
TS_ProcDescription 161
TS_ProcGroupIdentifier 159
TS_ProcGroupReference 158
TS_ProcId 154
TS_ProcId&ParList 155
TS_ProcIdentifier 156
TS_ProcResult 160
TS_TypeConstraints 501
TS_TypeDefs 69
TS_TypeIdentifier 738
TS_VarDcl 237
TS_VarDcls 233
TS_VarDclsGroup 230
TS_VarDclsGroupId 231
TS_VarDclsGroupIdentifier 232
TS_VarDclsOrGroup 229
TS_VarGroupIdentifier 236
TS_VarGroupRef 234
TS_VarGroupReference 235
TS_VarId 238
TS_VarIdentifier 239
TS_VarType 240
TS_VarValue 241
TTCN_ASP_Constraint 530
TTCN_ASP_ConstraintGroup 528
TTCN_ASP_ConstraintGroupId 529
TTCN_ASP_ConstraintOrGroup 527
TTCN_ASP_Constraints 526
TTCN_ASP_TypeDef 346
TTCN_ASP_TypeDefGroup 344
TTCN_ASP_TypeDefGroupId 345
TTCN_ASP_TypeDefOrGroup 343

TTCN_ASP_TypeDefs 342
TTCN_CM_Constraint 603
TTCN_CM_ConstraintGroup 601
TTCN_CM_ConstraintGroupId 602
TTCN_CM_ConstraintOrGroup 600
TTCN_CM_Constraints 599
TTCN_CM_TypeDef 422
TTCN_CM_TypeDefGroup 420
TTCN_CM_TypeDefGroupId 421
TTCN_CM_TypeDefOrGroup 419
TTCN_CM_TypeDefs 418
TTCN_Module 2
TTCN_ModuleExports 6
TTCN_ModuleId 3
TTCN_ModuleImportPart 20
TTCN_ModuleObjective 8
TTCN_ModuleOverviewPart 5
TTCN_ModuleRef 7
TTCN_ModuleStructure 19
TTCN_ObjectType 15
TTCN_PDU_Constraint 549
TTCN_PDU_ConstraintGroup 547
TTCN_PDU_ConstraintGroupId 548
TTCN_PDU_ConstraintOrGroup 546
TTCN_PDU_Constraints 545
TTCN_PDU_TypeDef 380
TTCN_PDU_TypeDefGroup 378
TTCN_PDU_TypeDefGroupId 379
TTCN_PDU_TypeDefOrGroup 377
TTCN_PDU_TypeDefs 376
TTCN_Specification 1
Type 734
Type&Attributes 396
Type&Restriction 81
TypeList 467
TypeOrPDU 733
TypeReference 131

U

UnaryOp 725
Underscore 772
Unit 303
UpperAlpha 760
UpperBound 402
UpperRangeBound 574
UpperTypeBound 88
UpperValueBound 585

V

ValRange 572
Value 739
ValueAttributes 578
ValueBound 582
ValueLength 580
ValueList 570
ValueRange 571
ValueReference 228
VarBlock 163
VarDcl 165
VarDcls 164
VariationDefault 474
VariationRef 472
VariationReference 473
VarIdentifier 167
VarIdentifiers 166
Verdict 674
VerdictId 673

W

WhileLoop 171
Wildcard 753

Annex B

Operational Semantics of TTCN

B.1 Introduction

Annex A describes the syntax of TTCN by means of BNF production rules and restrictions on these productions the observance of which may be verified either statically or dynamically.

This annex defines the semantics of TTCN by describing an abstract procedure that executes syntactically valid TTCN test suites. This procedure starts, for each Test Case, an abstract "TTCN machine" that evaluates this Test Cases by means of the creation, expansion and interpretation of an "EvaluationTree", dealing with one level (ordered set of alternatives in a certain position in the tree) at a time. In the execution of concurrent TTCN, additional TTCN machines are started, one for each created PTC. These machines work in the same way as the principal TTCN machine, which is then executing the main test component. The necessary PCOs and CPs, connecting TTCN machines with their environment and with each other, are assumed to exist already and to be initially empty.

The abstract procedure (EVALUATE_TEST_SUITE) and the TTCN machines (EVALUATE_TEST_CASE, EVALUATE_TEST_COMPONENT) are described in B.5. EvaluationTree has the form of a TTCN behaviour tree, but enriched by additional components. In a TTCN machine it is initially set to be the indicated Test Case or Test Step root tree, or local tree. In the course of test case execution, EvaluationTree is expanded, and "control" generally moves down the EvaluationTree, except in the execution of GOTOs and RETURNs, where control moves up.

The additional tree components, introduced for technical reasons, are the following: each node (alternative) has, besides the denoted StatementLine, a Boolean value IsDefault, telling whether the node stems from a Default Behaviour Table; each level has, besides the denoted list of StatementLines, a Boolean value IsExpanded, telling whether the level has already been expanded.

It is not required that a real TTCN machine be built in a way that it works internally exactly as the abstract one. TTCN operational semantics define only how a real TTCN machine should behave externally, i.e. with respect to PCO and CP queues, timers and the timer list, and test component termination information. Implementation details are irrelevant.

B.2 Precedence

Operational semantics for TTCN are supplied in the following subclauses in a mixture of pseudo-code and natural language. Where these two notations overlap they are meant to have identical meanings. If the pseudo-code and natural language conflict, this is an error, and should be reported back to the standards organization via a defect report. In such a case, pending correction of the defect by the standards organization, the pseudo-code will take precedence over the natural language text.

B.3 Processing of test case errors

Within the main body of this Recommendation, as well as within Annex A and this annex, conditions are described that result in the detection of test case errors. The observation of a test case error shall be recorded in the conformance log and lead to the abortion of the Test Case.

Without being explicitly mentioned in the following, a test case error is always detected dynamically if any part of an expression does not evaluate to a defined value. Expressions are evaluated, among other occasions, in the application of assignments, qualifiers, and constraints.

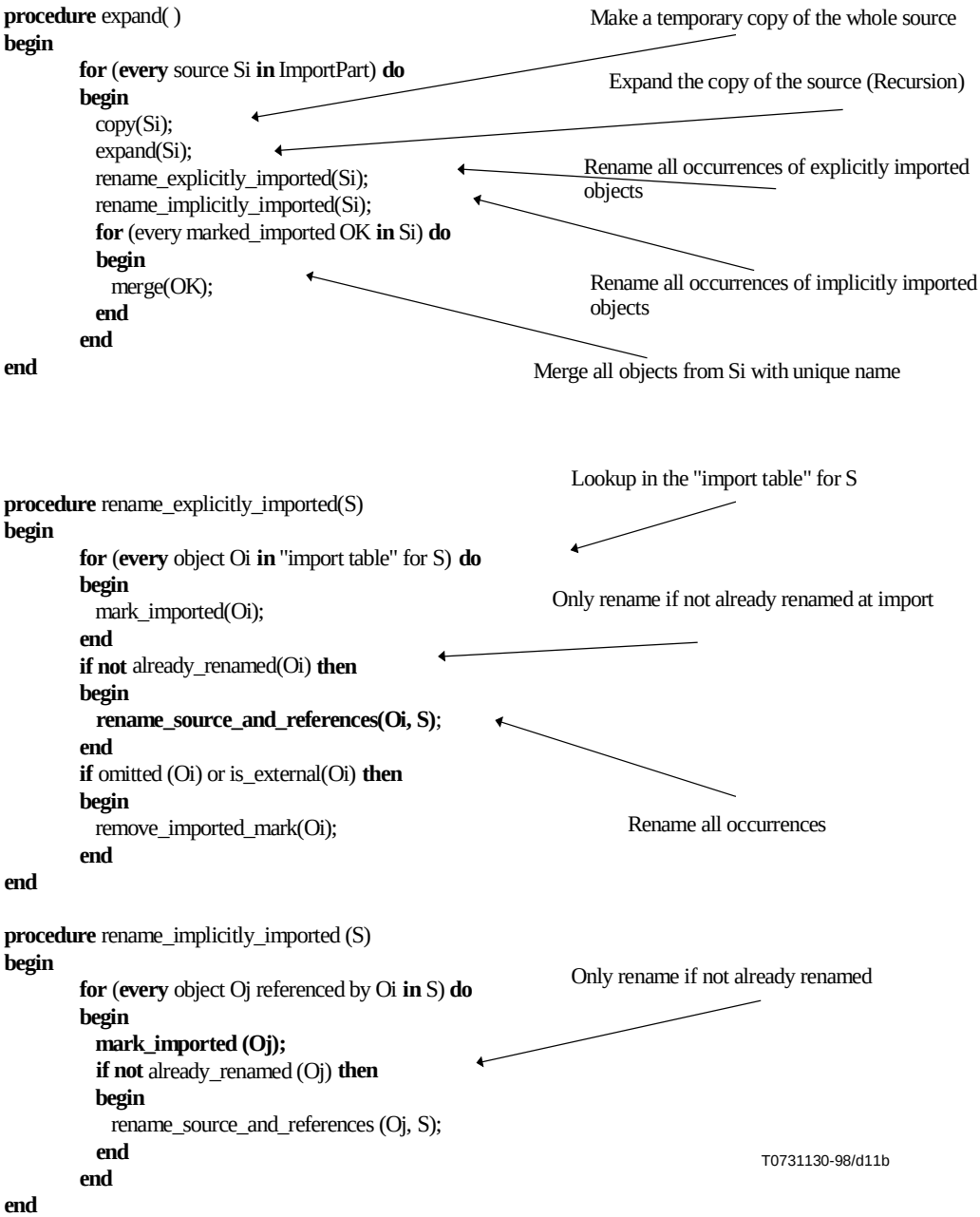
B.4 Converting a modularized test suite to an equivalent expanded test suite

This algorithm does not handle error cases. It requires that the objects are unique in the scope where they are defined and used.

In the conversion from modularized test suite to a expanded test suite, there is a need for the renaming of some imported TTCN objects (in order to avoid name clashes). In this rename process two options are allowed:

a) the original name is retained as defined in the declaration/definition of the object;

b) the new name is constructed by concatenation of the module identifier and the original name of the object. They shall be separated by two underscores, e.g. ModuleA_ConnectionRequest.



The principle of this algorithm is, for each source object, make a temporary copy of it, expand the copy, then mark each object to be imported and finally merge each marked object into the importing suite.

In expanding imported sources all explicitly and implicitly imported objects are renamed to Module::Identifier, if they were not already renamed at import. Every module shall have a unique identifier. In the expanded test suite all explicitly and implicitly imported objects are clearly recognizable and because every module has to have a unique name, name clashes are not possible.

B.5 TTCN operational semantics

B.5.1 Introduction

TTCN behaviour trees are evaluated one level of alternatives at a time. At each level, defaults are appended, attachment constructs are expanded, and REPEAT constructs are replaced. This produces a set of alternatives that can be evaluated to discover which one successfully matches and thereby determines which set of alternatives to proceed to next. The requirements for what constitutes a match for a TTCN statement depend on what is coded on that behaviour line, and are described in this semantics text.

B.5.2 The pseudo-code notation

B.5.2.1 Introduction

TTCN semantics are defined using a simple functional approach that explains the execution of a TTCN Test Case behaviour description, involving the step-wise expansion of an evaluation tree, and the execution of nodes of this tree. These functions are intended as an aid to understanding TTCN semantics and are not intended to be associated with any particular execution model or high level programming language. They are not meant to be direct methods for executing TTCN.

Keywords of pseudo-code are printed in bold font, e.g. **procedure**, **function**, **begin**, **end**, **if**, **then**, **else**. In the header of their definition, procedure, process, and function names are highlighted by bold font to facilitate lookup. For the same reason, the data type of a function is highlighted. Apart from this, data types are not dealt with explicitly.

B.5.2.2 Procedures and functions

Many statements are **procedure** calls. **Function** expressions may be used wherever a value of the associated type is needed. They obtain their value (and are immediately terminated) by **return**, followed by a value expression.

Procedure and function parameters are generally "throughput parameters", i.e. formal parameters that may be both "read" and "written to". In particular, functions may have "side effects" and are essentially "procedures with a value". Variables in a procedure or function body that are neither formal parameters nor any of the global ones mentioned above are local variables of this body, without explicit declaration.

Care is taken that:

– parameters are read only when they have a defined value;

– terms are used as actual parameters only where the procedure or function does not assign a value to the respective formal parameter, i.e. the parameter is purely an input parameter.

B.5.2.3 Processes

Processes behave like procedures, except that they are each run on a separate TTCN machine. They are not executed in a nested fashion. In a process, global data objects may be declared, such that they are available in all procedures and functions called in the process without being explicitly passed along as parameters. Avoiding long parameter lists makes the pseudocode easier to read. Of course, instances of global objects exist independently in each process (TTCN machine). There is no relationship between global objects in different processes.

In this annex, the following objects are treated as global objects in each process:

– EvaluationTree, of the Test Case (or Main Test Component) or Parallel Test Component;

– CurrentLevel, to be expanded or matched;

– Defaults, the current default context, used in default expansion;

– Snapshot, the temporarily fixed view of the environment;

– ReturnLevel, to be considered after the execution of a RETURN statement;

– ReturnDefaults, the default context of the ReturnLevel;

– SendObject, the ASP, PDU, or CM to be sent next;

– ReceiveObject, the ASP, PDU, or CM received last.

Thus, each TTCN machine will have its own EvaluationTree, etc.

Other objects, however, are accessible from all processes. The relevant state of the "environment of EVALUATE_TEST_SUITE", i.e. the contents of the relevant PCOs and CPs, as well as the lists of expired timers, the values of timers, and the list of terminated parallel test components, are assumed to be globally accessible from all test components and need not be passed explicitly as parameters. Similarly, Test Suite Parameters, Test Suite Constants, and Test Suite Variables are assumed to be accessible from all test case or test component processes.

B.5.2.4 Natural language within pseudo-code

Some parts of pseudo-code are written in natural language, in order to limit the complexity of this annex. These parts are enclosed by /# and #/. Such parts represent statements, for-loop details, or expressions of pseudo-code and are assumed to be executed or evaluated, when they are encountered.

Pure comments, intended for the human reader, not to be executed or evaluated by a TTCN machine, are enclosed by (* and *).

B.5.2.5 Levels and alternatives

A level visited in a tree denotes both a position in the tree and the ordered set of alternatives at this level.

An alternative visited in a tree determines a level position in the tree, see LEVEL_OF in B.5.25. The alternative denotes simultaneously a position in that level, a BehaviourLine, a StatementLine, etc.

Thus, levels and alternatives in a tree are pointers, but the unpacking of the data objects they point at is done implicitly.

B.5.3 Execution of a Test Suite

B.5.3.1 Introduction

The Test Suite is executed in the main procedure, EVALUATE_TEST_SUITE. Every Main Test Component (Test Case in the non-concurrent case) is executed on an abstract TTCN machine executing EVALUATE_TEST_CASE. Each Parallel Test Component is executed on an independent TTCN machine, performing EVALUATE_TEST_ COMPONENT.

```
procedure EVALUATE_TEST_SUITE(TestSuiteId)
        (* This procedure introduces unique names for all TTCN trees, including local subtrees. It sets Test Suite specific data
           objects and evaluates each Test Case whose selection expressions become TRUE. *)
begin
        for /# every Test Case, Test Step or Default behaviour table Table in TestSuiteId #/ do
        begin
          /#    Rename all local trees of Table such that they become unique throughout the test suite and different from any Test
                Case,
                Test Step or Default behaviour table name in the Test Suite. #/;
          /#    Rename accordingly in Table all references to local trees in attachments. #/;
          /#    Every node in every behaviour tree gets a new Boolean component "IsDefault".
                This component is set to TRUE for all nodes in Default Dynamic Behaviour Tables
                and FALSE for all nodes in all other tables. #/;
        end;
```

```
        for /# every Default behaviour table Table in TestSuiteId #/ do
        begin
          /#    For each leaf of the behaviour tree which does not have an entry in the verdict column assign the verdict
                R. #/
          /#    or each leaf of the behaviour table which has a preliminary result assigned, change the preliminary result to a
                verdict by removing the parentheses around it. #/
        end;
        Evaluated := /# empty list of Test Case Identifiers #/;
        /#  Set values of Test Suite Parameters, Test Suite Constants, and, where to be initialized, of Test Suite Variables #/;
        for /# every Test Case Identifier TCId of TestSuiteId that is not yet in Evaluated #/ do (* in any order *)
        begin
          SelEx := /# conjunction of the selection expressions of all test groups containing Test Case TCId (directly or via lower
          groups) #/;
          if EVALUATE_BOOLEAN(SelEx) then
                start process EVALUATE_TEST_CASE (TCId);
          /# add TCId to the list Evaluated #/;
        end
end
```

B.5.4 Execution of a Test Case

B.5.4.1 Execution of a Test Case – Pseudo-code

```
process EVALUATE_TEST_CASE(TestCaseId)
        (*  This process initializes the EvaluationTree by the Test Case root tree and the default context by the Defaults references
            listed with the Test Case Behaviour Description. It moves control to the top level of alternatives and calls their
            evaluation. *)
global EvaluationTree, CurrentLevel, Defaults, Snapshot, ReturnLevel, ReturnDefaults, SendObject, ReceiveObject;
begin
        /#  Initialize Test Case Variables, global R and MTC_R, PCOs, CPs, Timers, and the Timeout List of TestCaseId. #/;
        EvaluationTree := ROOT_TREE(TestCaseId);
          (*  EvaluationTree is a growing finite tree built up by pasting together and expanding copies of trees from the test case
              behaviour description and from the test step and default libraries. A component IsExpanded is added to each level. *)
          CurrentLevel := FIRST_LEVEL(EvaluationTree) ;
          (*  A level denotes both a position in a tree and the ordered set of alternatives at this position. *)
          ReturnLevel := CurrentLevel;
          Defaults := DEF_REF_LIST(TestCaseId);
          ReturnDefaults := Defaults;
          EVALUATE_LEVELS ();
          (*  This includes, by nested calls, the evaluation of all relevant subsequent levels in the growing evaluation tree. *)
end

procedure EVALUATE_LEVELS ()
        (*  This procedure first expands and evaluates CurrentLevel, which is the currently active level of alternatives of
            EvaluationTree. Defaults gives the currently active default context. The alternatives contained in CurrentLevel are
            processed in their order of appearance, if necessary in repeated rounds. CurrentAlternative is the loop variable of the for-
            loop, denoting the currently considered alternative in CurrentLevel. By the snapshot mechanism, in each round of
            matching attempts through CurrentLevel, the status of the environment considered does not change, giving each such
            round an instantaneous character.
            Save for dynamically detected test case errors, the evaluation of CurrentLevel includes the successful evaluation of an
            alternative. This is followed by the assignment of a verdict and the evaluation of the next level, and hence, by induction,
            of all levels that control subsequently moves to. *)
begin
          if NOT IS_EXPANDED() then
            (*  By this condition we avoid expanding levels repeatedly which are targets of GOTOs. *)
            EXPAND_CURRENT_LEVEL ();
          (*  Now the current level is free of REPEATs and attachments, and includes the necessary defaults. *)
          repeat
            (*  ... performing rounds through current level, trying to match an alternative. *)
            TAKE_SNAPSHOT();
            (*  ... of the incoming PCO and CP queue(s), the relevant timeout list, and the termination status of any other test
                components. *)
            for /# every CurrentAlternative in CurrentLevel, in the given order #/          do
            (*  try to match the current alternative. Note that an alternative visited in a tree determines a level position in the tree
                and denotes, depending on the context it is used in, a position in that level, a BehaviourLine, a StatementLine, etc. *)
```

```
    begin
      if EVALUATE_EVENT_LINE (CurrentAlternative) then
      (*  In the absence of Test Case errors the Test Component or Test Case will terminate inside the
          EVAL_VERDICT_ENTRY or GOTO_NEXT_LEVEL_OR_STOP_WITH_VERDICT call of the innermost
          recursive instance of EVALUATE_LEVELS, e.g. if there is a final verdict or no next level. Then, the for-loop will
          be aborted, too. *)
      begin
        if /# Alternative has a verdict column entry VerdictEntry #/ then
          EVAL_VERDICT_ENTRY(VerdictEntry);
        GOTO_NEXT_LEVEL_OR_STOP_WITH_VERDICT(CurrentAlternative);
        EVALUATE_LEVELS();
      end
    end
    until SNAPSHOT_FIXED();
    (*  SNAPSHOT_FIXED returns TRUE if Snapshot cannot change any more. *)
    LOG(TEST_CASE_ERROR);
    STOP_TEST_CASE();
end
```

B.5.4.2 Execution of a Test Case or Test Component – Natural language description

Step 1 Evaluation begins at the numerically lowest (in TTCN.MP), i.e. the leftmost (in TTCN.GR), level of indentation of the root tree.

Step 2 Expand current level to include all defaults explicitly, and to replace all tree attachments, as long as necessary, as well as all REPEATs, by their expansions.

Step 3 Take a snapshot of the incoming PCO and CP queue(s) and the timeout list.

NOTE 1 – The act of taking a snapshot does not remove an event from any PCO or CP.

Consider the first behaviour line at the current level of alternatives.

Step 4 Evaluate the TTCN statement on the current behaviour line.

The evaluation of each type of TTCN statement is specified in the operational semantics for that TTCN statement type.

Step 5 If the TTCN statement evaluates to a successful match, then go to Step 6.

Otherwise, if there are more alternatives in the current set of alternatives, consider the next behaviour line in the set of alternatives and go to Step 4.

If there are no more alternatives, and yet all PCO and CP queues relevant to this set of alternatives contain at least one event, and all timers relevant to Timeout statements in the set of alternatives are in the timeout list, then stop the Test Case and indicate *test case error*.

NOTE 2 – Under these conditions none of the set of alternatives can ever match.

In all other cases – i.e. there are no more alternatives and the next snapshot might show a different picture – go to Step 3.

Step 6 If a preliminary verdict is coded, process it as in B.5.23.2.

Step 7 If a leaf node in the tree or a node with a final verdict has been reached, then go to Step 8.

Otherwise, determine and consider the next level to be evaluated and go to Step 2.

Step 8 Use final verdict, or, if not specified, the current value of the preliminary result variable R, as the final verdict of the Test Case as in B.5.23.2 and B.5.25.

B.5.5 Expanding a set of alternatives

B.5.5.1 Introduction

This subclause defines how to expand a set of alternatives in preparation for evaluating which alternative matches.

This is done in four steps:

a) saving the Default context, if labelled level;

b) attachment of the current set of Default behaviour trees;

c) expansion of attached trees, if necessary, recursively, until there are no more attachment alternatives in the set;

d) expansion of REPEAT constructs, replacing them by a subtree in which tree attachments and GOTO constructs occur in lower levels only.

```
procedure EXPAND_CURRENT_LEVEL ()
begin
    if /# CurrentLevel has a label #/ then
      SAVE_DEFAULTS ();
    APPEND_DEFAULTS ();
    EXPAND_ATTACHMENTS (EvaluationTree, CurrentLevel, Defaults);
    (*   CurrentLevel is now free of tree attachments. *)
    EXPAND_REPEATS ();
    /# Component IsExpanded of CurrentLevel #/ := TRUE;
end
```

B.5.5.2 Saving Defaults

```
procedure SAVE_DEFAULTS ()
begin
        /# Replace CurrentLevel and its subsequent behaviour in the EvaluationTree by ACTIVATE (Defaults), followed by
CurrentLevel
        and its subsequent behaviour, with the label of the former CurrentLevel moved to the ACTIVATE line. #/;
        /# Consider new ACTIVATE line as the CurrentLevel #/;
end
```

B.5.5.3 Expansion of REPEAT constructs

If *RepeatedTree* denotes a particular TreeReference together with its ActualParList, and *Condition* denotes a particular Boolean expression, and *label* denotes a label not used anywhere else, then "REPEAT *RepeatedTree* UNTIL [*Condition*]" can be replaced by:

```
              [TRUE]
label          +RepeatedTree
                        [NOT (Condition)]
                            -> label
                        [Condition]
                            :
```

Lines describing subsequent behaviour of the REPEAT construct follow after [*Condition*] in this expansion, with an additional indentation of one level.

```
procedure EXPAND_REPEATS ()
begin
        for /# every alternative A in CurrentLevel, in the given order #/ do
        begin
          if /# A is of the form REPEAT RepeatedTree UNTIL [Condition] #/ then
          begin
            Subsequent := SUBSEQUENT_BEHAVIOUR_TO (EvaluationTree,A);
            Label := NEW_LABEL ();
            (*   Create a label which has been used neither in the (relabelled) Test Suite nor in the EvaluationTree. *)
            Expansion :=    MAKE_TREE ( "[TRUE]",
                        MAKE_TREE ( Label: "+" RepeatedTree,
                          MAKE_TREE ( "[NOT(" Condition ")]",
                                "->" Label,
                          MAKE_TREE ( "[" Condition "]",
                                    Subsequent,
                                    )),
                                ),
                            );
            REPLACE_ALT_TREE (EvaluationTree, CurrentLevel, A, Expansion);
          end
        end
end
```

B.5.5.4 Appending default behaviour

During evaluation of a test case, at each level of alternatives there is a current list of Default Tree References. This list comes either from the list in the appropriate Dynamic Behaviour Table, or from the most recently evaluated ACTIVATE construct. The appending of the Defaults is done by adding, for each entry in the current list of Defaults, the construct "+ DefaultReference" to the end of the set of alternatives.

```
procedure APPEND_DEFAULTS ()
begin
        for /# every D in Defaults, in the given order #/ do
        begin
          APPEND_TO_LEVEL (EvaluationTree, CurrentLevel, "+" D);
          (*  EvaluationTree and CurrentLevel are updated by appending the attachment of D to CurrentLevel. *)
        end
end
```

B.5.5.5 Expanding attached trees

Attached trees are expanded by replacing the attach construct + *TestStep* with the tree or, where applicable, the root tree of *TestStep* and subsequently, if there is behaviour specified following and indented from the Attach construct, to insert this behaviour after and indented from each leaf in the attached tree. Since attached trees may have their own list of default tree references in the header of the test step dynamic behaviour table, the expansion of tree attachment has to ensure that if any event on the first level of alternatives of the attached tree matches, then the defaults context is changed, and if a leaf node of that attached tree is reached without a verdict being assigned, then the defaults context of the calling tree is restored before the subsequent behaviour is evaluated. These changes in defaults context are most easily described in terms of the insertion of appropriate ACTIVATE constructs in the relevant places. If the attached tree is in fact a default tree, then there will be no default references in its header, so the ACTIVATE constructs that are inserted on entering that tree will have no parameters and thereby will deactivate all defaults within the scope of the default tree.

The attached trees on Level are expanded using the following procedure:

```
procedure EXPAND_ATTACHMENTS (Tree, Level, OuterDefaults)
begin
        for /# every alternative A in Level in Tree, in the given order #/ do
        begin
          if /# A is an ATTACH construct, i.e. of the form "+" AttachedTreeId ActualParList #/ then
          begin
            Subsequent := SUBSEQUENT_BEHAVIOUR_TO (Tree,A);
            AttachedTree := ROOT_TREE (AttachedTreeId);
            REPLACE_PARAMETERS (AttachedTreeId, AttachedTree, ActualParList);
            (*  This replaces the formal parameters in AttachedTree by the actual parameters specified in ActualParList, doing so by
                textual substitution *)
            RELABEL(AttachedTree);
            NewDefaults := DEF_REF_LIST(AttachedTreeId);
            NewLevel := FIRST_LEVEL(AttachedTree);
            EXPAND_ATTACHMENTS (AttachedTree, NewLevel, NewDefaults);
            EXPAND_SUBTREE (AttachedTree, Subsequent, NewDefaults, OuterDefaults);
            (* I.e.:  Insert ACTIVATE(NewDefaults) below first level of AttachedTree &
                      Attach ACTIVATE(OuterDefaults) and Subsequent to each leaf node of AttachedTree *)
            REPLACE_ALT_TREE(Tree, Level, A, AttachedTree);
          end
        end
end
```

```
procedure EXPAND_SUBTREE (SubTree, Subsequent, InnerDefaults, OuterDefaults)
        (*  This procedure first inserts ACTIVATE(InnerDefaults) below the first level of SubTree and then attaches
            ACTIVATE(OuterDefaults) and Subsequent to each leaf node of SubTree. *)
begin
        Level := FIRST_LEVEL(SubTree);
        for /# every alternative A of Level in SubTree #/ do
        begin
          SubOfA := SUBSEQUENT_BEHAVIOUR_TO (SubTree, A);
          ActTree := MAKE_TREE(A,
                MAKE_TREE("ACTIVATE(" InnerDefaults ")",
                    SubOfA, ), );
          REPLACE_ALT_TREE(SubTree, Level, A, ActTree);
        end
        for /# every leaf A in SubTree #/ do
        begin
          LeafTree :=   MAKE_TREE (A,
              MAKE_TREE ( "ACTIVATE(" OuterDefaults ")",
                    Subsequent, ), );
          REPLACE_ALT_TREE(SubTree, LEVEL_OF(SubTree, A), A, LeafTree);
        end
end
```

The expansion of attached trees is also explained in 15.13.

B.5.6 Evaluation of an Event Line

B.5.6.1 Pseudo-code

```
function EVALUATE_EVENT_LINE(Alternative): BOOLEAN
        (*  This function calls EVALUATE_EVENT, EVALUATE_PSEUDO_EVENT or EVALUATE_CONSTRUCT, depending
            on what type of StatementLine the current alternative is. *)
begin
        case    STATEMENT_LINE_TYPE_OF(Alternative) of
        begin
          EVENT:          if  EVALUATE_EVENT (Alternative)          then return TRUE; else return FALSE;
          PSEUDO_EVENT:   if  EVALUATE_PSEUDO_EVENT (Alternative)   then return TRUE; else return FALSE;
            CONSTRUCT:    (*  Construct can now only be GoTo, Return, Activate, Create. *)
                          if  EVALUATE_CONSTRUCT (Alternative)       then return TRUE; else return FALSE;
        end
end
```

B.5.6.2 Natural language description

Evaluate the TTCN statement on the current behaviour line, based on the statement type, i.e. whether it is an event, a pseudo-event, or a construct. The evaluation of each type of TTCN statement is specified in the operational semantics for that TTCN statement type in the following subclauses.

B.5.7 Functions for TTCN events

B.5.7.1 Functions for TTCN events – pseudo-code

```
function EVALUATE_EVENT(Alternative): BOOLEAN
        (*  This function calls SEND, RECEIVE, OTHERWISE, TIMEOUT , DONE, or IMPLICIT SEND, depending on what
            type of event the current alternative is. *)
begin
        case EVENT_TYPE_OF(Alternative) of
        begin
          SEND:           if  SEND (Alternative)          then return TRUE; else return FALSE;
          RECEIVE:        if  RECEIVE (Alternative)       then return TRUE; else return FALSE;
          OTHERWISE:      if  OTHERWISE (Alternative)     then return TRUE; else return FALSE;
          TIMEOUT:        if  TIMEOUT (Alternative)       then return TRUE; else return FALSE;
          DONE:           if  DONE (Alternative)          then return TRUE; else return FALSE;
          IMPLICIT_SEND:  if  IMPLICIT_SEND (Alternative) then return TRUE; else return FALSE;
        end
end
```

B.5.7.2 Functions for TTCN events – Natural language description

If the TTCN statement is an event, then it will be evaluated as specified in B.5.8 for a SEND event, B.5.9 for a RECEIVE event, B.5.10 for an OTHERWISE event, B.5.11 for a TIMEOUT event, B.5.12 for a DONE event, or B.5.13 for an IMPLICIT SEND event.

B.5.8 Execution of the SEND event

B.5.8.1 Execution of the SEND event – Pseudo-code

```
function SEND ( SendLine ): BOOLEAN
begin
        /# Read   PCOorCPidentifier,
                  ASPorPDUorCMidentifier,
                  Qualifier,
                  Assignments,
                  TimerOperations,
                  ConstraintsReference        from SendLine #/;
        if EVALUATE_BOOLEAN (Qualifier) then
        begin
          BUILD_SEND_OBJECT (ASPorPDUorCMidentifier, ConstraintsReference );
          EXECUTE_ASSIGNMENTS (Assignment);
          SEND_EVENT (PCOorCPidentifier, ConstraintReference);
          TIMER_OPS (TimerOperations);
          LOG(PCOorCPidentifier, SendObject);
          return TRUE;
        end
        else return FALSE;
end

procedure BUILD_SEND_OBJECT (ASPorPDUorCMidentifier, ConstraintsReference)
begin
        SendObject := /#   an instance of ASPorPDUorCMidentifier whose parameters/fields have the values specified by
                           ConstraintsReference #/;
end

procedure SEND_EVENT (PCOorCPidentifier, ConstraintsReference)
begin
        /#  Encode SendObject according to applicable encoding rules and variations,
            see ConstraintsReference and associated type definitions #/;
        /#  Put encoded SendObject at the end of OUTPUT_Q(PCOorCPidentifier) #/;
end
```

B.5.8.2 Execution of the SEND event - natural language description

The contents of the ASP or PDU or CM, as specified in the named Constraints Reference entry, are to be sent. Note that if there is a qualifier, the SEND can be executed only if that qualifier evaluates to TRUE.

Step 1 If there is a qualifier, then that qualifier will be evaluated before any other processing takes place:

- if the qualifier evaluates to FALSE, the SEND cannot succeed;
- if the qualifier evaluates to TRUE, then continue with Step 2.

Step 2 Create an ASP or PDU or CM as specified in the named Constraints Reference.

If the dynamic chaining feature has been used, then the value specified in the Constraints Reference entry will be assigned to the appropriate parameter or field of the ASP or PDU or CM to be sent.

Using the dynamic chaining feature has the effect of storing a copy of the named constraint into the named parameter or field of the ASP or PDU or CM being built for comparison. The structure defined for the associated Constraints Reference is used for this named parameter or field.

Step 3 If there is an Assignment statement, then that assignment will be performed as in B.5.16, in particular possibly changing the ASP or PDU or CM to be sent.

Step 4 The ASP or PDU or CM is now fully filled in according to the specifications given. The LT or UT will encode the PDUs (but not ASPs or CMs, apart from PDUs embedded in such) according to the applicable encoding rules. The LT or UT will send the ASP with its embedded encoded PDUs, or the encoded PDU. If a PCO or CP was stated, the ASP or PDU or CM is to be sent at that PCO or CP. If the PCO was not stated, i.e., the test uses a single PCO – then the ASP or PDU is sent from the lower PCO, because a CP cannot be implied.

Step 5 If one or more timer operations were coded on the behaviour line, the appropriate timer operation(s) will be performed as in B.5.17.

Step 6 Record in the conformance log the following information, as well as the information specified in B.5.24.2:

- the PCO or CP at which the SEND occurred;
- the fully defined ASP, PDU or CM that was sent.

B.5.9 Execution of the RECEIVE event

B.5.9.1 Execution of the RECEIVE event – Pseudo-code

```
function RECEIVE( ReceiveLine ): BOOLEAN
begin
        /# Read   PCOorCPidentifier,
                  ASPorPDUorCMidentifier,
                  Qualifier,
                  Assignments,
                  TimerOperations,
                  ConstraintsReference        from ReceiveLine #/;
        if /# INPUT_Q (PCOorCPidentifier) is not empty #/ then
        begin
          if ( OBJECT_MATCHES(PCOorCPidentifier, ASPorPDUorCMidentifier, ConstraintsReference)
              AND EVALUATE_BOOLEAN (Qualifier) ) then
          begin
              EXECUTE_ASSIGNMENTS (Assignments);
              TIMER_OPS (TimerOperations);
              REMOVE_OBJECT (PCOorCPidentifier);
              LOG(PCOorCPidentifier, ReceiveObject);
              return TRUE;
          end
          else return FALSE;
        end
        else return FALSE;
end

function OBJECT_MATCHES (PCOorCPidentifier, ASPorPDUorCMidentifier, ConstraintsReference): BOOLEAN
begin
        ReceiveObject := /# copy of encoded object at head of INPUT_Q(PCOorCPidentifier) #/;
        if /#  ReceiveObject can be decoded according to applicable encoding rules and variations,
               as given by ConstraintsReference and associated type definitions #/ then
        begin
          /# decode it, to yield new version of ReceiveObject #/;
          if  ( /# ReceiveObject is of type ASPorPDUorCMidentifier #/
                AND
                /# parameters/fields of ReceiveObject have values matching the ConstraintsReference #/ ) then
              return TRUE;
          else return FALSE;
        end
        else return FALSE;
end

procedure REMOVE_OBJECT (PCOorCPidentifier),
begin
    /# remove object at head of INPUT_Q(PCOorCPidentifier) #/;
end
```

B.5.9.2 Execution of the RECEIVE event – Natural language description

Step 1 If the snapshot that was taken when beginning the current iteration of checking this level of alternatives for matching shows that there is *no* incoming ASP or PDU or CM, then this RECEIVE cannot match.

Otherwise, continue to Step 2.

Step 2 If a PCO or CP was stated, the ASP or PDU or CM shall have been received at that PCO or CP. If the PCO was not stated, i.e., the test suite uses a single PCO – then the ASP or PDU shall have been received at the lower PCO. Note that a CP cannot be implied.

Step 3 The incoming PDUs are decoded according to the applicable encoding rules. A copy is made of the decoded incoming PDU or of the incoming ASP or CM with decoded nested PDUs.

Step 4 If the qualifier, possibly using values from the incoming data object, evaluates to FALSE, the RECEIVE cannot match. Otherwise, continue to step 5.

Step 5 A copy of the expected ASP or PDU or CM pattern is assembled, using the structure defined in the ASP or PDU or CM declaration plus the values, matching mechanisms and chained Constraints References specified in the named Constraints Reference.

This copy is compared against the incoming ASP or PDU or CM, and its decoded PDUs or the decoded PDU to determine if the RECEIVE can match as specified. Only if the RECEIVE did match successfully, continue to Step 6.

Step 6 The incoming ASP or PDU or CM which has just matched will be removed from the incoming PCO or CP queue and discarded.

Step 7 If there are Assignment statements, then they will be performed as in B.5.16.2.

Step 8 If one or more timer operations were coded on the behaviour line, the appropriate timer operation(s) will be performed as in B.5.17.

Step 9 Record in the conformance log the following information, as well as the information specified in B.5.24.2:

- the PCO or CP at which the RECEIVE occurred;
- the fully defined ASP, PDU or CM that was received.

B.5.10 Execution of the OTHERWISE event

B.5.10.1 Execution of the OTHERWISE event – Pseudo-code

```
function OTHERWISE ( OtherwiseLine): BOOLEAN
begin
        /# Read   PCOorCPidentifier,
                  Qualifier,
                  Assignments,
                  TimerOperations          from OtherwiseLine #/;
        if   ( /# INPUT_Q (PCOorCPidentifier) is not empty #/
             AND EVALUATE_BOOLEAN (Qualifier) ) then
        begin
          EXECUTE_ASSIGNMENTS (Assignments);
          TIMER_OPS (TimerOperations);
          REMOVE_OBJECT (PCOorCPidentifier);
          LOG(PCOidentifier, ReceivedObject);
          return TRUE;
        end
        else return FALSE;
end
```

B.5.10.2 Execution of the OTHERWISE event – Natural language description

The tester shall accept any incoming data that it has not been possible to decode or that has not matched a previous alternative to this OTHERWISE event. Note that if there is a qualifier, the OTHERWISE can only match if that qualifier evaluates to TRUE.

Step 1 If the qualifier evaluates to FALSE, the OTHERWISE cannot match. Otherwise, continue to step 2.

Step 2 If the snapshot that was taken when beginning the current iteration of checking this level of alternatives for matching shows that there is no incoming ASP, PDU or CM, then this OTHERWISE cannot match.

Otherwise, continue to Step 3.

Step 3 If a PCO was stated, the ASP or PDU shall have been received at that PCO. If a CP was stated, the CM shall have been received at that CP. If the PCO was not stated, i.e., the test uses a single PCO, then the ASP or PDU shall have been received at the lower PCO, because a CP cannot be implied.

Step 4 The incoming ASP, PDU or CM will be removed from the incoming PCO or CP queue and discarded.

Step 5 If there are Assignment statements, then they will be performed as in B.5.16.2.

Step 6 If one or more timer operations were coded on the behaviour line, the appropriate timer operation(s) will be performed as in B.5.17.

Step 7 Record in the conformance log the following information, as well as the information specified in B.5.24.2:

- the PCO or CP at which the OTHERWISE occurred;
- the ASP, PDU or CM that was received.

B.5.11 Execution of the TIMEOUT event

B.5.11.1 Execution of the TIMEOUT event – Pseudo-code

```
function TIMEOUT ( TimeoutLine ): BOOLEAN
begin
        /# Read   TimerIdentifier,
                  Qualifier,
                  Assignments,
                  TimerOperations          from TimeoutLine #/;
        if EVALUATE_BOOLEAN (Qualifier) then
        begin
          if TIMER_EXPIRED (TimerIdentifier) then
          begin
            EXECUTE_ASSIGNMENTS (Assignments);
            TIMER_OPS (TimerOperations);
            LOG(TimerIdentifier);
            return TRUE;
          end
          else return FALSE;
        end
        else return FALSE;
end
```

```
function TIMER_EXPIRED (TimerIdentifier): BOOLEAN
begin
        if /# TimerIdentifier is not empty #/ then
        begin
          if /# timeout notification from TimerIdentifier is in copy of timeout list in Snapshot #/ then
          begin
            /# delete timeout notification from TimerIdentifier in actual timeout list #/;
            /# stop and reset the timer TimerIdentifier #/;
          return TRUE;
          end
          else return FALSE;
        end
        else (* TimerIdentifier not specified *)
        begin
          if /# any timeout notification is in copy of timeout list in Snapshot #/ then
          begin
            /# stop and reset all timers mentioned in actual timeout list#/;
            /# delete all timeout notifications in actual timeout list #/;
            return TRUE;
          end
          else return FALSE;
        end
end
```

B.5.11.2 Execution of the TIMEOUT event – Natural language description

The tester will check to see if the named timer has expired. (If no timer name is given, the tester will check to see if *any* timer has expired.) Note that if there is a qualifier, the TIMEOUT is only considered as matching if that qualifier evaluates to TRUE.

Step 1 If there is a qualifier, then that qualifier will be evaluated before any other processing takes place.

- If the qualifier evaluates to FALSE, the TIMEOUT cannot match.
- If the qualifier evaluates to TRUE, then continue with Step 2.

Step 2 See if any of the timers explicitly or implicitly named on the TIMEOUT event have been running, but have expired.

- If no timer identifier is specified, then the tester shall check to see if *any* timer that had been running has now expired. If so, all timers which have timed out are reset (and left stopped). The timeout entry (entries) is (are) removed from the timeout list.
- If a timer identifier is specified, then the tester shall check to see if this timer had been running, but has now expired. If so, the expired timer is reset (and left stopped). The timeout entry is removed from the timeout list.
- If no timers have expired the TIMEOUT event can not match, i.e., the next alternative will be attempted.

Step 3 If there is an Assignment statement, then that assignment will be performed as in B.5.16.2.

Step 4 If one or more timer operations were coded on the behaviour line, the appropriate timer operation(s) will be performed as in B.5.17.

Step 5 Record in the conformance log the information specified in B.5.24, as well as the name of the timer that expired.

B.5.12 Execution of the DONE event

B.5.12.1 Execution of the DONE event – Pseudo-code

```
function DONE (DoneLine): BOOLEAN
begin
        /# Read   TCompList,
                  Qualifier,
                  Assignments,
                  TimerOperations          from DoneLine #/;
        if EVALUATE_BOOLEAN (Qualifier) AND     ALL_TERMINATED(TCompList) then
        begin
          EXECUTE_ASSIGNMENTS (Assignments);
          TIMER_OPS (TimerOperations);
          LOG(TCompList);
          return TRUE;
        end
        else return FALSE;
end

function ALL_TERMINATED(TCompList): BOOLEAN
begin
        if   TCompList =/# EmptyList #/ then
             TCompList := /# list of all created Parallel Test Components #/;
        for /# every TComp in TCompList #/ do
        begin
          if /# TComp has not terminated in the Snapshot #/  then
            return FALSE;
        end
        return TRUE;
end
```

B.5.12.2 Execution of the DONE event – Natural language description

The termination status of the given list of Test Components is to be checked. If all given components have terminated (at the time of the last SNAPSHOT), then the event matches, provided that the qualifier also evaluates to TRUE.

Step 1 If there is a qualifier, then that qualifier will be evaluated before any other processing takes place:

- if the qualifier evaluates to FALSE, the DONE cannot succeed;
- if the qualifier evaluates to TRUE, the continue to Step 2.

Step 2 If all test components listed in TCompList had terminated at the time of the last SNAPSHOT, then continue to Step 3, otherwise this DONE cannot match.

Step 3 If there is an Assignment statement, then that assignment will be performed as in B.5.16.

Step 4 If one or more timer operations were coded on the behaviour line, the appropriate timer operation(s) will be performed as in B.5.17.

Step 5 Record in the conformance log the information specified in B.5.24, as well as the TCompList.

B.5.13 Execution of the IMPLICIT SEND event

B.5.13.1 Execution of the IMPLICIT SEND event – Pseudo-code

```
function IMPLICIT_SEND (Alternative): BOOLEAN
begin
        /# Execute IMPLICIT_SEND according to natural language description #/;
        return TRUE;
end
```

B.5.13.2 Execution of IMPLICIT SEND – Natural language description

The IUT is induced to do whatever is necessary to send the contents of the ASP or PDU, as specified in the constraints reference entry of the alternative.

If the dynamic chaining feature has been used, then the value specified in the Constraints Reference entry will be assigned to the appropriate parameter or field of the ASP or PDU to be sent.

IMPLICIT SENDing always succeeds.

B.5.14 Execution of a pseudo-event

B.5.14.1 Execution of a pseudo-event – Pseudo-code

```
function EVALUATE_PSEUDO_EVENT ( PseudoEventLine ): BOOLEAN
begin
        /# Read   Qualifier,
                  Assignments,
                  TimerOperations          from PseudoEventLine #/;
        if EVALUATE_BOOLEAN (Qualifier) then
        begin
          EXECUTE_ASSIGNMENTS (Assignments);
          TIMER_OPS (TimerOperations);
          LOG( );
          return TRUE;
        end
        else return FALSE;
end
```

B.5.14.2 Execution of PSEUDO-EVENTS – Natural language description

If the TTCN statement is a pseudo-event, then it will be evaluated as specified in B.5.15 for a Boolean Expression, B.5.16 for an Assignment Statement, B.5.17 for a timer operation (START, CANCEL, or READTIMER).

After completion of the pseudo-event, record in the conformance log the information specified in B.5.24.

B.5.15 Execution of BOOLEAN expressions

B.5.15.1 Execution of BOOLEAN expressions – Pseudo-code

```
function EVALUATE_BOOLEAN(Qualifier): BOOLEAN
begin
        if /# Qualifier is empty #/ then
          return TRUE;
        else
        begin
          if /# Qualifier evaluates to TRUE #/ then
            return TRUE;
          else return FALSE;
        end
end
```

B.5.15.2 Execution of BOOLEAN expressions – Natural language description

A Boolean expression (i.e., qualifier) specifies a condition that is to be tested. This condition will either be TRUE or FALSE. A Boolean expression may be stated as part of a statement line (i.e., on the same line with a SEND, RECEIVE, TIMEOUT, or OTHERWISE), or as a statement line on its own (i.e., as a pseudo-event).

Step 1 The Boolean expression shall be evaluated to determine if the condition specified is TRUE or FALSE. The normal rules of Boolean Logic apply, with the precedence rules specified in 11.4.2.1.

B.5.16 Execution of assignments

B.5.16.1 Execution of assignments – Pseudo-code

```
procedure EXECUTE_ASSIGNMENTS (AssignmentList)
begin
        for /# every assignment CurrentAssignment in AssignmentList, in the given order #/ do
        begin
          /# Execute CurrentAssignment #/;
        end
end
```

B.5.16.2 Execution of ASSIGNMENTs – Natural language description

The assignment list is evaluated in left to right order. In each assignment, the variable on the left-hand side of that statement is to take on the value of the expression on the right-hand side of the statement. This expression is evaluated observing the precedence indicated in Table 3.

If the assignment is performed in a Send line, the left-hand side may denote an ASP-, PDU- or CM-component, referring to the object to be sent. If the assignment is performed in a Receive line, the expression may refer to components of the ASP-, PDU- or CM to be received.

B.5.17 Execution of TIMER operations

B.5.17.1 Execution of TIMER operations – Pseudo-code

```
procedure TIMER_OPS (TimerOperations)
begin
        for /# every TimerOperation in TimerOperations #/ do
        case TIMER_OP_TYPE_OF(TimerOperation) of
        begin
          START_TIMER:          START_TIMER(TimerOperation);
          CANCEL_TIMER:         CANCEL_TIMER(TimerOperation);
          READ_TIMER:           READ_TIMER(TimerOperation);
        end
end

procedure START_TIMER (TimerOperation)
begin
        /# perform as in B.5.17.2 #/;
end

procedure CANCEL_TIMER (TimerOperation)
begin
        /# perform as in B.5.17.3 #/;
end

procedure READ_TIMER (TimerOperation)
begin
        /# perform as in B.5.17.4 #/;
end
```

B.5.17.2 Execution of START_TIMER – Natural language description

Step 1 If the timer is already running, cancel it and continue to Step 2. Otherwise continue directly to Step 2.

Step 2 The timer is to be started with an initial value indicating no time has passed. Any entry for this timer in the timeout list is removed from the list.

B.5.17.3 Execution of CANCEL_TIMER – Natural language description

The CANCEL_TIMER operation specifies that a timer (or timers) is to stop ticking.

Step 1 Determine the name of the timer(s) to be cancelled:

- if no timer identifier is specified, then cancel *all* timers;
- if a timer identifier is specified, then cancel the timer with this timer identifier.

Step 2 The status of the named or implied timer(s) is to be set to "not running". The amount of time elapsed for the timer(s) is to be set to zero. If the timeout list contains an entry for the timer(s), the entry (entries) is (are) removed from the list.

B.5.17.4 Execution of READ_TIMER – Natural language description

The READ_TIMER operation specifies that the amount of time that has passed for a currently running timer is to be stored into a variable. The timer continues to run without interruption.

Step 1 Interrogate the value of the timer having the specified name. If the amount of time passed is *n* of the units declared for this timer type, store *n* into the named variable.

If the timer is not currently running, the named variable shall be set to zero.

B.5.18 Functions for TTCN constructs

B.5.18.1 Functions for TTCN constructs – Pseudo-code

```
function EVALUATE_CONSTRUCT (Construct): BOOLEAN
        (* As the EvaluationTree is expanded at the CurrentLevel, the REPEAT and ATTACH constructs are not encountered here. *)
begin
        case CONSTRUCT_TYPE_OF(Construct) of
        begin
          ACTIVATE:     ACTIVATE(Construct);
          CREATE:       CREATE (Construct);
          GOTO:         (* no action here, see GOTO_NEXT_LEVEL_OR_STOP_WITH_VERDICT *);
          RETURN:       (* no action here, see GOTO_NEXT_LEVEL_OR_STOP_WITH_VERDICT *);
        end
        return TRUE;
end
```

B.5.18.2 Functions for TTCN constructs – Natural language description

If the TTCN statement is a TTCN construct, then it will be evaluated as specified in B.5.19 for an ACTIVATE construct, as specified in B.5.20 for a CREATE construct, as specified in B.5.21 for a GOTO construct, or as specified in B.5.22 for a RETURN construct. There is no need to deal with REPEATs, as they all have been replaced in the CurrentLevel.

TTCN constructs will always succeed.

B.5.19 Execution of the ACTIVATE construct

B.5.19.1 Execution of the ACTIVATE construct – Pseudo-code

```
procedure ACTIVATE ( ActivateLine )
begin
    /#  Read   DefRefList   from ActivateLine #/;
    Defaults:=DefRefList;
    LOG(DefRefList);
end
```

B.5.19.2 Execution of the ACTIVATE construct – Natural language description

Change the current defaults context to the DefaultRefList that appears as parameter to the ACTIVATE construct.

Step 1 Change default context to DefaultRefList.

Step 2 Record in the conformance log the following information as well as the information specified in B.5.24:

- the DefaultRefList.

B.5.20 Execution of the CREATE construct

B.5.20.1 Execution of the CREATE event – Pseudo-code

```
procedure CREATE ( CreateLine ): BOOLEAN
begin
        /#  Read   CreateList    from CreateLine #/;
        for /# every (TCompIdentifier,   TreeReference,   ActualParList) drawn from CreateList #/ do
        begin
          start process EVALUATE_TEST_COMPONENT(TCompIdentifier, TreeReference, ActualParList);
          (*  This starts the concurrent evaluation of TreeReference. *)
          LOG(TCompIdentifier,TreeReference, ActualParList);
        end
end

process EVALUATE_TEST_COMPONENT(TCompId, TreeReference, ActualParList)
        (*  This process initializes the EvaluationTree by the appropriate Test Step root tree or local tree and the default context by
            the Defaults references listed with the corresponding behaviour table. It moves control to the top level of alternatives and
            calls their evaluation. *)
global EvaluationTree, CurrentLevel, Defaults, Snapshot, ReturnLevel, ReturnDefaults, SendObject, ReceiveObject;
begin
    /#  Initialize the local instances of Test Case Variables, local R, Timers, and the Timeout List of TCompId. #/;
    EvaluationTree := ROOT_TREE(TreeReference);
      (*  EvaluationTree is a growing finite tree built up by pasting together and expanding copies of trees from the test case
          behaviour description and from the test step and default libraries. A component IsExpanded is added to each level. *)
    REPLACE_PARAMETERS (TreeReference, EvaluationTree, ActualParList);
      CurrentLevel := FIRST_LEVEL(EvaluationTree) ;
      (*  A level denotes both a position in a tree and the ordered set of alternatives at this position. *)
    ReturnLevel := CurrentLevel;
    Defaults := DEF_REF_LIST(TreeReference);
    ReturnDefaults := Defaults;
    EVALUATE_LEVELS ();
    (*  This includes, by nested calls, the evaluation of all relevant subsequent levels in the growing evaluation tree. *)
end
```

B.5.20.2 Execution of the CREATE event – Natural language description

The evaluation of the given Test Component is to be started.

Step 1 Evaluation of TCompIdentifier, bound to TreeReference, is started, with the ActualParList parameters replacing the Formal Parameters by textual substitution in TreeReference. All Test Case Variables, the local result variable R, timers and the local timeout list are provided afresh for the sole use by this test component.

Step 2 Record in the conformance log the following information as well as the information specified in B.5.24:

- the TCompIdentifier;
- the TreeReference;
- the ActualParList.

B.5.21 Execution of the GOTO construct

Control is transferred to the set of alternatives having the specified target label in the labels column. Execution now continues at this new level.

In pseudo-code, the GOTO construct is performed as a part of GOTO_NEXT_LEVEL_OR_STOP_WITH_VERDICT.

B.5.22 Execution of the RETURN construct

Control is transferred to the set of alternatives from which the defaults were entered the last time. Execution now continues at this new level.

In pseudo-code, the RETURN construct is performed as a part of GOTO_NEXT_LEVEL_OR_STOP_WITH_VERDICT.

B.5.23 The verdict

B.5.23.1 The verdict – Pseudo-code

```
procedure EVAL_VERDICT_ENTRY (VerdictEntry)
begin
        /# Expand VerdictEntry to full word, e.g. (P) becomes (PASS) #/;
        if /# VerdictEntry is a preliminary verdict "("PrelimVerdict")" #/ then
        begin
          UPDATE_PRELIM ( PrelimVerdict, /# local R, or MTC_R in case of Main Test Component #/);
          UPDATE_PRELIM ( PrelimVerdict, /# global R #/);
        end
        else (*  VerdictEntry is a final verdict. *)
        begin
          if /# Current process is EVALUATE_TEST_CASE #/ then
          begin
            EXCLUDE_INCOMPATIBLE_ENTRY ( VerdictEntry, /# global R #/);
            LOG(VerdictEntry);
            /# assign final verdict in main test component or test case #/;
            TERMINATE_TEST_CASE();
          end
          else (*  Process is EVALUATE_TEST_COMPONENT *)
          begin
            EXCLUDE_INCOMPATIBLE_ENTRY ( VerdictEntry, /# local R #/);
            UPDATE_PRELIM          ( VerdictEntry, /# global R #/);
            stop process;
          end
        end
end

process EXCLUDE_INCOMPATIBLE_ENTRY (Entry, RVal)
begin
        if (  ( Entry = "R" AND /# RVal = none #/ ) OR
              (Entry = "PASS" AND /# Rval = inconc #/ ) OR
              (Entry = "PASS" AND /# Rval = fail #/ ) OR
              (Entry = "INCONC" AND /# Rval = fail #/ ) ) then
        begin
          LOG(TestCaseError);
          STOP_TEST_CASE();
          return FALSE;
        end
        else return TRUE;
end

procedure UPDATE_PRELIM (PrelimVerdict, ResultVar)
begin
        if (  ResultVar = none OR
              (ResultVar = pass AND PrelimVerdict <> PASS) OR
              (ResultVar = inconc AND PrelimVerdict = FAIL) ) then
        begin
          /# replace value of ResultVar by PrelimVerdict in lower case letters #/;
          LOG("("PrelimVerdict")");
        end
end
```

B.5.23.2 The VERDICT – Natural language description

If a verdict is coded, process the verdict:

- If the verdict is preliminary, i.e. enclosed in parentheses, then the local and global result variables will be updated according to the verdict algorithm in 15.17.2. Note that in the Main Test Component the local R is denoted by MTC_R. The stated verdict is recorded in the conformance log.

- If the verdict is R, then, in non-concurrent TTCN or in the Main Test Component, the current value of R (the only or the global R) will be used as the verdict of the Test Case. If R is set to none, raise a test case error.
- If the verdict is PASS, INCONC or FAIL, then, in non-concurrent TTCN or in the Main Test Component, the stated verdict will be used as the final verdict for the Test Case. If the final verdict is inconsistent with local or global R, raise a TestCaseError.
- In Parallel Test Components, a final verdict R, PASS, INCONC or FAIL, is used to update the global R like a preliminary verdict. The stated verdict is recorded in the conformance log. A final verdict terminates the evaluation of the Test Component.

B.5.24 The Conformance Log

B.5.24.1 The LOG – Pseudo-code

```
procedure LOG( /# any number of arguments #/ )
begin
        /# log the line number of the event line (if any) #/;
        /# log the label associated with the event line (if any) #/;
        /# log the arguments passed to LOG #/;
        /# log the assignment(s) made (if any) #/;
        /# log the timer operation(s) performed (if any) #/;
        /# log current time #/; (*  current time may be actual or relative *)
end
```

B.5.24.2 The conformance log – Natural language description

Record the following information in the conformance log:

- the line number of the event line (if any);
- the label associated with the event line (if any);
- other arguments defined elsewhere in this annex associated with the event line (if any), e.g. the final or preliminary verdict, or the data object sent or received;
- the assignment(s) made (if any);
- the timer operation(s) performed (if any);
- time stamp.

B.5.25 Tree handling functions and procedures

To facilitate lookup, the procedures and functions are defined in alphabetical order.

```
procedure APPEND_TO_LEVEL (Tree,Level,Alternative)
begin
        /# Update Level and Tree by appending Alternative as new last alternative in Level in Tree #/;
end

function FIRST_LEVEL (Tree): LEVEL
begin
        return  /# the set of alternatives at the first level of indentation of Tree, i.e. the numerically lowest (in TTCN.MP),
                i.e. the leftmost (in TTCN.GR), level of indentation of the root tree #/;
end
```

```
procedure GOTO_NEXT_LEVEL_OR_STOP_WITH_VERDICT(Alternative)
begin
        (*  search the next level to evaluate, if any *)
        if /# Alternative is of the type "GOTO Label" or "-> Label" #/ then
          CurrentLevel := /# the unique level labelled with Label #/;
        else if /# Alternative is of the type "RETURN" #/ then
        begin
          CurrentLevel := ReturnLevel;
          Defaults := ReturnDefaults;
        end
        else if /# Alternative is a leaf of EvaluationTree #/; (*  but not a RETURN or GOTO *) then
          EVAL_VERDICT_ENTRY("R"); (*  This will stop the execution of the process. *)
        else
          CurrentLevel := /# set of alternatives at next level of indentation below Alternative #/;
        (*  save information for coming RETURN statements *)
        if /# Component IsDefault of CurrentLevel #/ = FALSE then
        begin
          ReturnLevel := CurrentLevel;
          ReturnDefault := Default;
        end
end

function IS_EXPANDED (): BOOLEAN
begin
        return /# Component IsExpanded of CurrentLevel #/;
end

function LEVEL_OF (Tree, Alternative): LEVEL
begin
        return /# the level in Tree of which this Alternative is a member #/;
end

function MAKE_TREE (Statement, Tree1, Tree2): TREE
begin
        return /# the following tree:
            Statement
              Tree1
            Tree2                                #/   ;
        (*  Tree1 and/or Tree2 may be empty, denoted by an empty parameter position in the call of MAKE_TREE. *)
end

function NEW_LABEL ( ): LABEL
begin
        return /# a label which has not yet been used in the execution of this Test Component, nor in the (relabelled) Test Suite #/ ;
              (*  This may be achieved by means of counters and test component names. *)
end

procedure RELABEL (Tree)
begin
        for /# each label L originally occurring in Tree #/ do
          begin
            NewLabel := NEW_LABEL();
            for /# each occurrence of L in Tree, in the label column or as the target of a GOTO #/ do
              begin
                /# replace L by NewLabel #/;
              end
          end
end

procedure REPLACE_ALT_TREE (Tree, Level, A, ReplacementTree)
begin
        (*  A is an alternative in Level, which is a level in Tree *)
        /#  In Tree, replace the subtree of Tree consisting of
            A and SUBSEQUENT_BEHAVIOUR_TO (Tree, A) by ReplacementTree,
            with all values of IsDefault in ReplacementTree set to the IsDefault-value of A,
            and all values of IsExpanded of levels in ReplacementTree set to FALSE. #/;
end
```

```
procedure REPLACE_PARAMETERS (TreeId, Tree, ActualParList)
begin
        /#  Replace the formal parameters in Tree by the actual parameters specified in ActualParList,
            doing so by textual substitution in Tree, using the formal parameter list accessible via TreeId. #/;
end

function ROOT_TREE (TreeId): TREE
begin
        return  /#  its root tree if TreeId denotes a Test Case or Test Step or Default Behaviour Table –
                    otherwise the local tree with this name. Each level gets a new Boolean component
                    "IsExpanded", initialized with value FALSE, indicating that this level has not yet been expanded. #/;
end

function SUBSEQUENT_BEHAVIOUR_TO (Tree, Alternative): TREE
begin
        return /# the subtree below Alternative in Tree #/;
          (*  This would be Tree3 if Tree has the form:
              Tree1
                Tree2
                Alternative
                  Tree3
                Tree4
              Tree5                    *)
end
```

B.5.26 Miscellaneous functions used by the pseudo-code

```
function CONSTRUCT_TYPE_OF(Construct): CONSTRUCT_TYPE
begin
        return /# ACTIVATE, CREATE, GOTO, or RETURN, as appropriate #/;
end

function DEF_REF_LIST(TreeReference): DEFAULT_REF_LIST
begin
        return /# the default reference list in the header of the corresponding table in the case of a test step in the test step library, or
      the empty list in the case of default behaviour, or in the case of a local tree attachment the current value of Defaults (i.e. the
      currently active defaults in the calling tree) #/;
end

function EVENT_TYPE_OF(Alternative): EVENT_TYPE
begin
        return /# SEND, RECEIVE, OTHERWISE, TIMEOUT, DONE, or IMPLICIT_SEND, as appropriate #/;
end

function INPUT_Q(PCOorCPidentifier): QUEUE
begin
        if /# PCOorCPidentifier is empty #/ then
          return /# default PCO input queue #/;
        else return /# input queue identified by PCOorCPidentifier #/;
end

function OUTPUT_Q(PCOorCPidentifier): QUEUE
begin
        if /# PCOorCPidentifier is empty #/ then
          return /# default PCO output queue #/;
        else return /# output queue identified by PCOorCPidentifier #/;
end

function SNAPSHOT_FIXED (): BOOLEAN
begin
        if /# all relevant PCO and CP queue(s) have some event(s) on them and all relevant timers have expired #/ then
          return TRUE;
        else return FALSE;
end

function STATEMENT_LINE_TYPE_OF(Alternative): STATEMENT_LINE_TYPE
begin
        return /# EVENT, PSEUDO_EVENT, or CONSTRUCT, as appropriate #/;
end
```

```
procedure STOP_TEST_CASE()
begin
        /# stop all running processes #/;
end

procedure TAKE_SNAPSHOT()
        (*  A snapshot of the incoming PCO and CP queue(s), the relevant timeout list, and the termination status of any other test
            components is taken. The act of taking a snapshot does not remove an event from any PCO, CP or timeout list.*)
begin
        /# save current PCO and CP input queues in Snapshot #/;
        /# save current timeout list in Snapshot #/;
        /# save current list of terminated Test Components in Snapshot #/;
end

procedure TERMINATE_TEST_CASE()
begin
        if /# any Parallel Test Component processes are still running #/ then
          LOG(TEST_CASE_ERROR);
        STOP_TEST_CASE();
end

function TIMER_OP_TYPE_OF(Alternative): TIMER_OP_TYPE
begin
        return /# START_TIMER, CANCEL_TIMER, or READ_TIMER, as appropriate #/;
end
```

Annex C

TTCN Modules

C.1 Introduction

A TTCN Module shall contain the following sections in the order indicated:

a) TTCN Module Overview Part;

b) Import Part;

c) Declarations Part;

d) Constraints Part;

e) Dynamic Part.

C.2 TTCN Module Overview Part

C.2.1 Introduction

The purpose of the TTCN Module Overview Part of a module is to provide information needed for the use of the module by other modules or test suites. This includes:

a) TTCN Module Exports;

b) TTCN Module Structure;

c) Test Case Index;

d) Test Step Index;

e) Default Index.

C.2.2 TTCN Module Exports

The TTCN Module Exports proforma identifies the module and provides information on the overall objective of the TTCN Module (e.g. constraints library for a particular protocol).

If a PCO type is given as an exported object in the Export table, it must be defined in the optional PCO Type table.

The name of the original source object shall be given if the object is imported.

If the object is declared as an external object (explicit external) or is an object which is omitted in the imported source object (implicit external), the keyword EXTERNAL is given instead of the source object name.

Exporting an object of type Enumeration or Named Number requires that the corresponding type is given. The other objects which are defined in the corresponding type are not exported as well. They are however implicitly exported and can be referred in other exported objects. The type name is given as a suffix to the object name embedded in brackets.

The following information shall be supplied in the TTCN Module Exports:

a) the name of the TTCN Module;

b) a description of the objective of the module;

c) a full reference of the TTCN module;

d) references to the relevant base standards if any;

e) a reference to the PICS proforma if any;

f) a reference to the PIXIT proforma if any;

g) an indication of the test method(s) if any;

h) other information which may aid understanding of the TTCN Module; this should be included as a comment;

i) a list of exported objects,

   where the following information shall be supplied for each exported object:

   1) The name of the object.

      If the object is of type NamedNumber or Enumeration, the corresponding type shall be given as a suffix to the object name embedded in brackets.

   2) The object type.

   3) The name of the original source object if the object is imported, or the object directive EXTERNAL.

   4) A page number,

      providing the location of the object in the module (no page number shall be given for imported objects).

This information shall be provided in the format shown in Proforma C.1, below.

TTCN Module Exports				
TTCN Module Name : *TTCN_ModuleIdentifier*				
Objective : *[FreeText]*				
TTCN Module Ref : *[FreeText]*				
Standards Ref : *[FreeText]*				
PICS Ref : *[FreeText]*				
PIXIT Ref : *[FreeText]*				
Test Method(s) : *[FreeText]*				
Comments : *[FreeText]*				
Object Name	**Object Type**	**Source Name**	**Page No.**	**Comments**
⋮ *ObjectIdentifier* ⋮	⋮ *TTCN_ObjectType* ⋮	⋮ *[SourceIdentifier| ObjectDirective]* ⋮	⋮ *Number* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*				

Proforma C.1 – TTCN Module Exports

SYNTAX DEFINITION:

4 TTCN_ModuleIdentifier ::= Identifier

12 ObjectIdentifier ::= Identifier | ObjectTypeReference

15 TTCN_ObjectType ::= **SimpleType_Object | StructType_Object | ASN1_Type_Object | TS_Op_Object | TS_Proc_Object | TS_Par_Object | SelectExpr_Object | TS_Const_Object | TS_Var_Object | TC_Var_Object | PCO_Type_Object | PCO_Object | CP_Object | Timer_Object | TComp_Object | TCompConfig_Object | TTCN_ASP_Type_Object | ASN1_ASP_Type_Object | TTCN_PDU_Type_Object | ASN1_PDU_Type_Object | TTCN_CM_Type_Object | ASN1_CM_Type_Object | EncodingRule_Object | EncodingVariation_Object | InvalidFieldEncoding_Object | Alias_Object | StructTypeConstraint_Object | ASN1_TypeConstraint_Object | TTCN_ASP_Constraint_Object | ASN1_ASP_Constraint_Object | TTCN_PDU_constraint_Object | ASN1_PDU_Constraint_Object | TTCN_CM_Constraint_Object | ASN1_CM_Constraint_Object | TestCase_Object | TestStep_Object | Default_Object | NamedNumber_Object | Enumeration_Object**

17 SourceIdentifier ::= SuiteIdentifier | TTCN_ModuleIdentifier

18 ObjectDirective ::= Omit | **EXTERNAL**

EXAMPLE C.1 – TTCN Module Exports:

TTCN Module Exports

TTCN Module Name	**:**	*TTCN_Module_A*
Objective	**:**	To illustrate the use of the TTCN Module Exports table
TTCN Module Ref	**:**	
Standards Ref	**:**	
PICS Ref	**:**	
PIXIT Ref	**:**	
Test Method(s)	**:**	
Comments	**:**	

Object Name	Object Type	Source Name	Page No.	Comments
String5	SimpleType_Object		3	
wait	Timer_Object	Module_B		
INTC	TTCN_PDU_Type_Object		13	
DEF1	Default_Object	TestSuite_1		
TC_2	TestCase_Object	TestSuite_2		
TC_3	TestCase_Object		33	
Preamble	TestStep_Object	EXTERNAL		

C.2.3 TTCN Module Structure

The TTCN Module Structure contains a list of Test Groups in the module (if any). The following information shall be supplied for each group:

a) the Test Group Reference,

   where the first identifier may be the module name, and each successive identifier represents further conceptual ordering of the module;

b) an optional selection expression identifier;

c) the Test Group Objective;

d) a page number (page number shall not be supplied for imported groups).

This information shall be provided in the format shown in Proforma C.2, below.

TTCN Module Structure			
Test Group Reference	**TestGroupReference**	**Test Group Objective**	**Page No.**
⋮ *TestGroupReference* ⋮	⋮ *TestGroupReference* ⋮	⋮ *FreeText* ⋮	⋮ *Number* ⋮
Detailed Comments: *[FreeText]*			

Proforma C.2 – TTCN Module Structure

SYNTAX DEFINITION:

626 TestGroupReference ::= [SuiteIdentifier "/"] {TestGroupIdentifier "/"}

The static semantics described in 10.2 "Test Suite Structure" are applicable for TTCN Module Structure.

C.2.4 Test Case Index

The definition of the Test Case Index for modules is the same as the definition of Test Case Index for Test Suites.

C.2.5 Test Step Index

The definition of the Test Step Index for modules is the same as the definition of Test Step Index for Test Suites.

C.2.6 Default Index

The definition of the Default Index for modules is the same as the definition of Default Index for Test Suites.

C.3 Import Part

C.3.1 Introduction

The purpose of the Import Part of a module is to declare the objects which are not explicitly defined but have been used. These objects are either declared as external objects or are imported from other source objects. This part includes:

a) External;

b) Import.

C.3.2 External

The External Objects proforma lists the objects being referred to by their identifier in the TTCN module, but neither imported nor explicitly defined. An external object lets the importer know what he has to define, when importing the TTCN module.

The following information shall be supplied for each external object:

a) the Object identifier and parameters,

   parameters are included when the object is a Test Suite Operation, a Constraint or a Test Step;

b) the object type;

c) an optional comment.

This information shall be provided in the format shown in Proforma C.3, below.

External Objects		
Object Name	**Object Type**	**Comments**
⋮ *Identifier	TS_OpId&ParList |* *ConsId&ParList	TestStepId&ParList* ⋮	⋮ *TTCN_ObjectType* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*		

Proforma C.3 – External Objects

SYNTAX DEFINITION:

141 TS_OpId&ParList ::= TS_OpIdentifier [FormalParList]

555 ConsEk&ParList ::= ConstraintIdentifier [FormalParList]

638 TestStepId&ParList ::=TestStepIdentifier [FormalParList]

15 TTCN_ObjectType ::= **SimpleType_Object | StructType_Object | ASN1_Type_Object | TS_Op_Object | TS_Proc_Object | TS_Par_Object | SelectExpr_Object | TS_Const_Object | TS_Var_Object | TC_Var_Object | PCO_Type_Object | PCO_Object | CP_Object | Timer_Object | TComp_Object | TCompConfig_Object | TTCN_ASP_Type_Object | ASN1_ASP_Type_Object | TTCN_PDU_Type_Object | ASN1_PDU_Type_Object | TTCN_CM_Type_Object | ASN1_CM_Type_Object | EncodingRule_Object | EncodingVariation_Object | InvalidFieldEncoding_Object | Alias_Object| StructTypeConstraint_Object | ASN1_TypeConstraint_Object | TTCN_ASP_Constraint_Object | ASN1_ASP_Constraint_Object | TTCN_PDU_constraint_Object | ASN1_PDU_Constraint_Object | TTCN_CM_Constraint_Object | ASN1_CM_Constraint_Object | TestCase_Object | TestStep_Object | Default_Object | NamedNumber_Object | Enumeration_Object**

EXAMPLE C.2 – External Objects:

External Objects		
Object Name	**Object Type**	**Comments**
CRC(P:A_PDU) CONSTRAINT_A(acstr:t_CONNECT) TESTSTEP_A(I:INTEGER) DEF3	TS_Op_Object TTCN_PDU_Constraint_Object TestStep_Object Default_Object	

C.3.3 Import

The definition of the Import for modules is the same as the definition of Import for Test Suites (see 10.7).

Annex D

Test Suite Index

D.1 Introduction

The Test Suite Index is a complete list of all objects in a expanded test suite and is a result of converting a modularized test suite to an expanded test suite. This list contains information about each object (e.g. the source object/test suite name, the original name and the page number in the very original source object).

D.2 The Test Suite Index

D.2.1 Introduction

The purpose of the Test Suite Index is to provide information needed for all imported objects in an expanded test suite. This information is used to easily find the definition of an object.

D.2.2 The Test Suite Index

The Test Suite Index proforma identifies all objects used in a test suite. The following information shall be supplied for each object:

a) the name of the object,

   the name with which the object is referred to (e.g. a generated name);

b) the object type,

   which shall be the same as the type given when the object is defined;

c) the name of the source object or the test suite,

   where the object is defined;

d) the original name of the object,

   the given name when the object is explicitly defined;

e) an optional page number,

   providing the location of the object in the original source object.

This information shall be provided in the format shown in Proforma D.1, below.

Test Suite Index					
Object Name	**Object Type**	**Source Name**	**Original Object Ref**	**Page No.**	**Comments**
⋮ *ObjectIdentifier* ⋮	⋮ *ObjectType* ⋮	⋮ *SourceIdentifier* ⋮	⋮ *[ObjectReference]* ⋮	⋮ *[Number]* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments: *[FreeText]*					

Proforma D.1 – Test Suite Index

The page number is given when the original source object is Recommendation and the location of the object is unambiguous.

Annex E

Compact proformas

E.1 Introduction

As an option, many Constraints and/or many Test Cases can be printed in a single table. This may be useful to highlight relations between the single constraints and/or single Test Cases. This annex states the requirements for using compact Constraints proformas and/or compact Test Cases proformas and gives some examples. These proformas are specific and differ from the generalized layouts given in 7.3. Since the new proformas are only another way to present the same information, there is no TTCN.MP associated with it. The information contained in a compact Constraints and/or compact Test Cases table can be translated in the TTCN.MP associated with the many single constraint tables and/or many Test Case tables that have the same information contents.

E.2 Compact proformas for constraints

E.2.1 Requirements

It shall only be allowed to print many single constraint tables as a single compact constraint table if:

a) the constraints have the same ASP type, PDU type, Structured Type or ASN.1 Type;

b) there is no encoding information specified in any of the single constraint table headers, nor in the encoding column of any of those tables (ASN.1 encodings specified in ASN.1 Value may, however, be specified in compact proformas); and

c) there are no entries in the comments column of any single constraint table.

NOTE – If the single constraints tables only have comments in the detailed comments footer (i.e. the comments column is empty), then it is possible to print these constraints in the compact format. In such cases, the individual detailed comments from the single proformas should be collected and printed as a single comment in the detailed comments footer of the compact proforma.

E.2.2 Compact proformas for ASP constraints

In cases where a constraint contains only a few parameters, or when there are only a small number of constraints, the constraints may be presented in the compact version of the ASP constraints Proforma E.1, below.

ASP Constraints Declarations					
ASP Type : *StructIdentifier*					
Constraint Name	**Derivation path**	**Field Name**			**Comments**
		ASP_ParIdentifier$_1$		*ASP_ParIdentifier*$_n$	
ConsId-&ParList$_1$	*Derivation-Path*$_1$	*ConstraintValue-&Attributes*$_{1,1}$		*ConstraintValue-&Attributes*$_{1,n}$	*[FreeText]*$_1$
ConsId-&ParList$_2$	*Derivation-Path*$_2$	*ConstraintValue-&Attributes*$_{2,1}$		*ConstraintValue-&Attributes*$_{2,n}$	*[FreeText]*$_2$
⋮	⋮	⋮		⋮	⋮
ConsId-&ParList$_m$	*Derivation-Path*$_m$	*ConstraintValue-&Attributes*$_{m,1}$		*ConstraintValue-&Attributes*$_{m,n}$	*[FreeText]*$_m$

Proforma E.1 – (Compact) ASP Constraints Declarations

This proforma is used for ASPs and their parameters in the same way that the PDU Constraints Declarations proforma is used for PDUs and their fields (see E.2.3).

E.2.3 Compact proformas for PDU constraints

E.2.3.1 Introduction

In cases where a constraint contains only a few fields, or when there are only a small number of constraints, the constraints may be presented in the compact version of the PDU constraints Proforma E.2, below.

The compact constraints proforma has field names across the top of the proforma, and different instances of the PDU constraints in rows within the proforma. If there are *n* fields in the PDU type definition, then there shall be *n* field columns in the compact constraint proforma.

The derivation path column is optional; however, it shall be used to specify the derivation path of modified constraints (see 13.6). A compact table can collect several base constraints (as illustrated in Example C.1) or can collect a base constraint and its modified constraints as in Example C.2. When modified constraints are declared in a compact table, the fields not modified in the modified constraints appear as boxes left blank at the intersection of the modified constraint row and of the field column. When mapping a compact table to TTCN.MP (i.e. single format), blank fields due to inheritance shall be omitted. Fields not specified in modified constraints are left blank in modified constraints.

PDU Constraints Declarations					
PDU Type : *PDU_Identifier*					
Constraint Name	**Derivation path**	**Field Name**			**Comments**
		ASP_ParIdentifier$_1$		*ASP_ParIdentifier*$_n$	
ConsId-&ParList$_1$	*Derivation-Path*$_1$	*ConstraintValue-&Attributes*$_{1,1}$		*ConstraintValue-&Attributes*$_{1,n}$	*[FreeText]*$_1$
ConsId-&ParList$_2$	*Derivation-Path*$_2$	*ConstraintValue-&Attributes*$_{2,1}$		*ConstraintValue-&Attributes*$_{2,n}$	*[FreeText]*$_2$
⋮	⋮	⋮		⋮	⋮
ConsId-&ParList$_m$	*Derivation-Path*$_m$	*ConstraintValue-&Attributes*$_{m,1}$		*ConstraintValue-&Attributes*$_{m,n}$	*[FreeText]*$_m$

Proforma E.2 – (Compact) PDU Constraints Declarations

EXAMPLE E.1 – Constraints using the compact constraints proforma

E.1.1 Given the declaration of PDU_B to be:

External Objects		
PDU Name : PDU_B		
PCO Type : XSAP		
Comments :		
Field Name	**Field Value**	**Comments**
FIELD1	INTEGER	
FIELD2	BOOLEAN	
FIELD3	IA5String	

E.1.2 The constraints on PDU_B using the compact constraints proforma could be:

PDU Constraints Declarations				
PDU Type : PDU_B				
Constraint Name	**Field Name**			**Comments**
	FIELD1	FIELD2	FIELD3	
CN1	3	TRUE	"A string"	
CN2	(4,5,6)	FALSE	"A string"	
CN3	0	?	–	

The constraints reference in the dynamic part might then contain entries such as PDU_B[CN1] and PDU_B[CN2].

E.1.3 The inheritance mechanism using the compact constraint proforma:

PDU Constraints Declarations						
PDU Type : PDU_A						
Constraint Name	**Derivation Path**	**Field Name**				**Comments**
		FIELD1	FIELD2	FIELD3	FIELD2	
CN0		0	'FF'H	'00'B	TRUE	
CN1	CN0	1				
CN2	CN0. CN		–	?		

E.2.3.2 Parameterized compact constraints

Compact constraints may also be parameterized. In such cases, the parameter lists shall be appended to the constraint name and occur in the constraint name column of compact constraint proformas.

EXAMPLE E.2 – A parameterized compact constraint

The invocation of the constraints on PDU_X in a Test Step may be made as follows: S1, S2, S3, S4, S5(0), S5(1) or S5(Var) where Var is a Test Case or Test Suite Variable:

PDU Constraints Declarations			
PDU Type : PDU_X			
Constraint Name	**Field Name**		**Comments**
	P1	P2	
S1	0	0	
S2	0	1	
S3	1	0	
S4	1	1	
S5(A:INTEGER)	1	A	

E.2.4 Compact proformas for Structured Type constraints

Compact Structured Type constraints shall be provided in Proforma E.3, below.

Structured Type Constraints Declarations					
Structured Type : *StructIdentifier*					
Constraint Name	**Derivation path**	**Field Name**			**Comments**
		ASP_ParIdentifier$_1$		*ASP_ParIdentifier*$_n$	
ConsId-&ParList$_1$	*Derivation-Path*$_1$	*ConstraintValue-&Attributes*$_{1,1}$		*ConstraintValue-&Attributes*$_{1,n}$	*[FreeText]*$_1$
ConsId-&ParList$_2$	*Derivation-Path*$_2$	*ConstraintValue-&Attributes*$_{2,1}$		*ConstraintValue-&Attributes*$_{2,n}$	*[FreeText]*$_2$
⋮	⋮	⋮		⋮	⋮
ConsId-&ParList$_m$	*Derivation-Path*$_m$	*ConstraintValue-&Attributes*$_{m,1}$		*ConstraintValue-&Attributes*$_{m,n}$	*[FreeText]*$_m$

Proforma E.3 – (Compact) Structured Type Constraints Declarations

EXAMPLE E.3 – Use of structured compact constraints

The PDU_Y consists of five fields named Y1 through Y5. The fields Y1, Y2 and Y3 have been combined into the Structured Type called A. In the following, the first table shows the constraints defined on PDU_Y. The second and third tables convey the same information as the last table.

The second and third tables show the Structured Type A's constraint specification using the single constraint proformas, while the last table shows A's constraint using the compact constraint proforma. Both figures also use the modification mechanism.

For the following tables, it can be seen that if the constraint YY1 was used, the values for field Y1 through Y5 would be 0,0,0,0,1 respectively, where the values for fields Y1 through Y3 are derived from the Structured Type A using constraint A1. If the constraint YY2 was used, the values for Y1 through Y5 would be 0,3,0,1,0 respectively, where the values for fields Y1 through Y3 are derived from the Structured Type A using constraint A2.

E.3.1 A PDU constraints table that uses a Structured Type (called A):

PDU Constraints Declarations				
PDU Type : PDU_Y				
Constraint Name	**Field Name**			**Comments**
	A	Y4	Y5	
YY1	A1	0	1	
YY2	A2	1	0	
YY3	A2	0	1	

E.3.2 A1 is a base constraint of Structured Type A:

Structured Type Constraint Declaration		
Constraint Name : A1		
Structured Type : A		
Derivation Path :		
Comments :		
Element Name	**Element Value**	**Comments**
Y1	0	
Y2	0	
Y3	0	

E.3.3 The Structured Type constraint, A2, is a modified constraint derived from A1:

Structured Type Constraint Declaration		
Constraint Name : A2		
Structured Type : A		
Derivation Path : A1		
Comments :		
Element Name	**Element Value**	**Comments**
Y2	3	

E.3.4 Structured Type A's constraints A1 and A2 in the compact form:

Structured Type Constraint Declaration					
Structured Type Name : A					
Constraint Name	**Derivation Path**	**Field Name**			**Comments**
		A	Y4	Y5	
A1 A2	 A1	0	0 3	0	

When using Structured Types within PDU Constraint Declarations, each field name used within the Structured Type definition shall exactly match the name (or short name, if both the short name and full name were defined) of the PDU field which it represents from the original PDU type definition.

E.2.5 Compact proformas for ASN.1 constraints

Proformas E.4, E.5 and E.6 shall be used for compact ASN.1 ASP, ASN.1 PDU and ASN.1 Type constraints definitions respectively:

ASN.1 ASP Constraints Declarations	
ASP Type : *ASP_Identifier*	
Constraint name	**ASN.1 Value**
Consld&ParList$_1$	*ConstraintValue&Attributes*$_1$
. . .	. . .
Consld&ParList$_m$	*ConstraintValue&Attributes*$_m$

Proforma E.4 – (Compact) ASN.1 ASP Constraints Declarations

ASN.1 PDU Constraints Declarations	
PDU Type : *ASP_Identifier*	
Constraint name	**ASN.1 Value**
Consld&ParList$_1$	*ConstraintValue&Attributes*$_1$
. . .	. . .
Consld&ParList$_m$	*ConstraintValue&Attributes*$_m$

Proforma E.5 – (Compact) ASN.1 PDU Constraints Declarations

ASN.1 Type Constraints Declarations	
Type Name : *ASP_Identifier*	
Constraint name	**ASN.1 Value**
ConsId&ParList$_1$	*ConstraintValue&Attributes*$_1$
.	.
.	.
.	.
ConsId&ParList$_m$	*ConstraintValue&Attributes*$_m$

Proforma E.6 – (Compact) ASN.1 Type Constraints Declarations

E.3 Compact proforma for Test Cases

E.3.1 Requirements

It is only permitted to print many single Test Case dynamic behaviour tables as a single compact Test Case dynamic behaviour table when the following rules apply:

a) all single Test Case dynamic behaviour tables shall belong to the same Test Group;

b) all single Test Case dynamic behaviour tables shall have either the same Default tree or no Default tree; it is recommended that there be no Default tree;

c) the behaviour description of each single Test Case dynamic behaviour table shall consist of a single ATTACH construct.

E.3.2 Compact proforma for Test Case dynamic behaviours

Where a series of Test Cases have essentially the same dynamic behaviour and differences occur only in the referenced constraints (e.g. tests for parameter variations of ASPs and/or PDUs), the Test Cases may be presented in the compact version of the Test Case dynamic behaviour Proforma E.7, below.

Test Case Dynamic Behaviours			
Group : *TestGroupReference* **Default** : *DefaultReference*			
Test Case Name	**Purpose**	**Test Step Attachment**	**Comments**
⋮ *TestCaseIdentifier* ⋮	⋮ *FreeText* ⋮	⋮ *Attach* ⋮	⋮ *[FreeText]* ⋮
Detailed Comments:			

Proforma E.7 – (Compact) Test Case Dynamic Behaviours

Each row in the body of this proforma describes a single Test Case. If the compact Test Case proforma is used, the single table replaces a series of Test Case dynamic behaviour tables in the behaviour part of the test suite.

The comments column contains comments pertaining to individual Test Cases against each attachment.

Test Cases within compact Test Case proforma may form a subset of their group and shall appear in the order indicated in the Test Case Index.

EXAMPLE E.4 – A compact Test Case table that defines a series of tests for FTAM:

Test Case Dynamic Behaviours		
Group : R/BV/PV/LM/CR/OV		
Default :		
Test Case Name	**Purpose**	**Test Step Attachment**
OVERRIDE1	Omit the override parameter, when file exists.	+OVERRIDE (FCRERQ_001,FCRERP_001)
OVERRIDE2	Omit the override parameter, when file does not exist.	+OVERRIDE (FCRERQ_002,FCRERP_002)

Appendix I

Examples

I.1 Examples of tabular constraints

I.1.1 ASP and PDU definitions

I.1.1.1 Flat Type definition

PDU Type Definition		
PDU Name : R/BV/PV/LM/CR/OV		
PCO Type :		
Comments : Illustration of TTCN mechanisms		
Field Name	**Field Value**	**Comments**
Source	BITSTRING [4]	Length is 4 bits.
Destination	BITSTRING [4]	Length is 4 bits.
T_Class	INTEGER0to4	Defined as a simple type
UserData	IA5String	

I.1.1.2 Structured Type definition

PDU Type Definition		
PDU Name : R/BV/PV/LM/CR/OV		
PCO Type :		
Comments : Illustration of TTCN mechanisms		
Field Name	**Field Type**	**Comments**
T_Addresses	T_AddressInfo	
T_Class	INTEGER0to4	Defined as a simple type
UserData	IA5String	

Structured Type Definition		
Type Name : T_CONNECT2		
Comments : Can be used in all Transport PDU examples.		
Element Name	**Type Definition**	**Comments**
Source Destination	BITSTRING[4] BITSTRING[4]	Length is 4 bits Length is 4 bits

I.1.1.3 Special Type PDU, in order to allow use of (static) chaining of constraints

ASP Type Definition		
ASP Name : N_DATArequest		
PCO Type : N_SAP		
Comments : For illustration only		
Parameter Name	**Parameter Type**	**Comments**
CallingNetworkAddress CalledNetworkAddress ConnectionIdentifier Data	HEXSTRING HEXSTRING HEXSTRING PDU	 To enable chaining of constraints

I.1.2 ASP/PDU constraints

I.1.2.1 Flat

PDU Constraint Declaration		
Constraint Name : TCON_CLASS4_1		
PDU Type : T_CONNECT1		
Derivation Path :		
Comments :		
Field Name	**Field Value**	**Comments**
Source Destination T_Class UserData	TS_Par1 TS_Par2 4 "testing, testing"	

I.1.2.2 Structured, referring to field groups

PDU Constraint Declaration		
Constraint Name : TCON_CLASS4_2		
PDU Type : T_CONNECT2		
Derivation Path :		
Comments :		
Field Name	**Field Value**	**Comments**
T_Addresses T_Class UserData	WrongAddress 4 "one, two, three"	WrongAddress is a reference to a structured type constraint.

Structured Type Constraint Declaration		
Constraint Name : WrongAddress		
Structured Type : T_AddressInfo		
Derivation Path :		
Comments :		
Element Name	**Element Value**	**Comments**
Source Destination	TS_Par1 '0000'B	

I.1.2.3 Chaining, useful for (nested) PDUs in ASPs

ASP Constraint Declaration		
Constraint Name : N_DATAreq_With_T_CON_Class4_1		
ASP Type : N_DATArequest		
Derivation Path :		
Comments : TCON_Class4_1 is a PDU constraint (i.e. chaining).		
Parameter Name	**Parameter Value**	**Comments**
CallingNetworkAddress CalledNetworkAddress ConnectionIdentifier Data	TS_Par3 TS_Par4 'ABCDEF'H TCON_Class4_1	

I.1.2.4 Parameterized constraints: it is possible to parameterize flat, structured and chained constraints. The following example shows parameterization to pass a value:

PDU Constraint Declaration		
Constraint Name : TCON_1(class:INTEGER)		
PDU Type : T_CONNECT1		
Derivation Path :		
Comments : TCON_Class4_1 is a PDU constraint (i.e. chaining).		
Field Name	**Field Value**	**Comments**
Source Destination T_Class UserData	'1000'B ? class ?	 Class is a formal parameter.

This can be referenced from the Test Case, Test Step or Default behaviour tables, as for example:

TCON_1(4) or TCON_1(TCvariable)

Field values may be whole (chained) PDUs:

ASP Constraint Declaration		
Constraint Name : N_DATAreq_With_T_CON(A_Constraint:T_CONNECT2)		
ASP Type : N_DATArequest		
Derivation Path :		
Comments : TCON_Class4_1 is a PDU constraint (i.e. chaining).		
Parameter Name	**Parameter Value**	**Comments**
CallingNetworkAddress	TS_Par3	
CalledNetworkAddress	TS_Par4	
ConnectionIdentifier	'1234567'H	
Data	A_Constraint	A_Constraint is a formal parameter.

This constraint can be called, as for example:

N_DATAreq_With_TCON(TCON_Class4_2)

Since the actual parameter is a constraint name, which can itself be parameterized, it is possible to express an arbitrary depth of nesting of PDUs.

I.1.2.5 Modified constraints: it is possible to use existing constraints and modify them to define new constraints. This can be done with flat, structured and parameterized constraints:

PDU Constraint Declaration		
Constraint Name : TCON_CLASS0_1		
PDU Type : T_CONNECT1		
Derivation Path : TCON_Class4_1		
Comments : Class 0 is acceptable.		
Field Name	**Field Value**	**Comments**
T_Class	0	

Wildcards can be used for values:

PDU Constraint Declaration		
Constraint Name : TCON_AnyClass		
PDU Type : T_CONNECT1		
Derivation Path : TCON_Class4_1		
Comments : Any class (0..4) is acceptable.		
Field Name	**Field Value**	**Comments**
T_Class	?	

This is considered to be bad style, however. It is better to use the more general constraint as a base.

It is also possible to delete whole fields:

PDU Constraint Declaration		
Constraint Name : TCON_Erroneous_NoClass		
PDU Type : T_CONNECT1		
Derivation Path : TCON_Class4_1		
Comments : No class present		
Field Name	**Field Value**	**Comments**
T_Class	–	T_Class omitted

I.2 Examples of ASN1 constraints

I.2.1 ASP and PDU definitions

I.2.1.1 Flat

ASN.1 PDU Type Definition		
PDU Name	**:**	T_CONNECT1
PCO Type	**:**	
Comments	**:**	
Type Definition		
-- only to illustrate use of ASN.1 in TTCN		
SEQUENCE {	source	BITSTRING (SIZE (4..4)),
	Destination	BITSTRING (SIZE (4..4)),
	t_Class	INTEGER (0..4)
	userData	IA5String OPTIONAL
}		

I.2.1.2 Structured

ASN.1 PDU Type Definition		
PDU Name	**:**	T_CONNECT2
PCO Type	**:**	
Comments	**:**	
Type Definition		
-- only to illustrate use of ASN.1 in TTCN		
SEQUENCE {	t_Addresses	T_AddressInfo
	t_Class	INTEGER (0..4)
	userData	IA5String
}		
-- expansion of T_AddressInfo can be found in a table of its own.		

Related ASN.1 productions that are normally in one ASN.1 module may be distributed over more tables in TTCN:

ASN.1 Type Definition		
Type Name	**:**	T_AddressInfo
Comments	**:**	
Type Definition		
SEQUENCE {	source	BITSTRING (SIZE (4..4)),
	destination	BITSTRING (SIZE (4..4)),
}		

I.2.1.3 An ASP definition

ASN.1 ASP Type Definition				
ASP Name	**:** N_DATArequest			
PCO Type	**:** N_SAP			
Comments	**:**			
Type Definition				
SEQUENCE	{	callingNetworkAddress	OCTETSTRING,	-- *even number of octets*
		calledNetworkAddress	OCTETSTRING,	-- *even number of octets*
		connectionIdentifier	OCTETSTRING,	-- *even number of octets*
		Data	T_PDUS	
	}			

ASN.1 Type Definition		
Type Name	**:** T_PDUS	
Comments	**:**	
Type Definition		
CHOICE	{	t1 T_CONNECT1 t2 T_CONNECT2 }

I.2.2 ASN.1 ASP/PDU constraints

I.2.2.1 Flat

ASN.1 PDU Constraint Declaration			
Constraint Name	**:** TCON_Class4_1		
PDU Type	**:** T_CONNECT1		
Derivation Path	**:** T_CONNECT1		
Comments	**:**		
Constraint Value			
{	source	TS_PAR1,	
		TS_PAR2,	-- *field identifier can be omitted if desired.*
	t_Class	4	
	userData	"testing, testing"	
}			

I.2.2.2 Structured

ASN.1 PDU Constraint Declaration	
Constraint Name	**:** TCON_Class4_2
PDU Type	**:** T_CONNECT2
Derivation Path	**:**
Comments	**:**

Constraint Value

```
{ t_Addresses      WrongAddress,    -- a reference to a PDU field constraint
  t_Class          4,
  userData         "one, two, three"
}
```

ASN.1 PDU Constraint Declaration	
Constraint Name	**:** WrongAddress
PDU Type	**:** T_AddressInfo
Derivation Path	**:**
Comments	**:**

Constraint Value

```
{ source      TS_PAR1,
  destination      '0000'B
}
```

I.2.2.3 Chaining a PDU constraint

ASN.1 ASP Constraint Declaration	
Constraint Name	**:** N_DATAreq_With_TCON_Class4_1
ASP Type	**:** N_DATArequest
Derivation Path	**:**
Comments	**:**

Constraint Value

```
{ callingNetworkAddress     TS_PAR_3
  calledNetworkAddress      TS_PAR_4
  connectionIdentifier      'ABCDEF'H
  data                      t1  TCON_Class4_1  -- chaining to a PDU constraint
}
```

I.2.2.4 Parameterized constraints: ASN.1 constraints may be parameterized like TTCN tabular constraints, for example:

ASN.1 PDU Constraint Declaration		
Constraint Name	**:**	TCON_1(class:INTEGER)
PDU Type	**:**	T_CONNECT1
Derivation Path	**:**	
Comments	**:**	

Constraint Value			
{	source	’0000’B,	
	destination	?,	-- *wildcard*
	t_Class	class,	-- *formal parameter*
	userData	?	
}			

This can be referenced from the Test Case, Test Step or Default behaviour tables, as for example:

TCON_1(4) or TCON_1(TCvariable)

A parameter may also represent a whole chained PDU:

ASN.1 ASP Constraint Declaration		
Constraint Name	**:**	N_DATAreq_With_TCON(a_constraint:T_CONNECT2)
ASP Type	**:**	N_DATArequest
Derivation Path	**:**	
Comments	**:**	

Constraint Value		
{	callingNetworkAddress	TS_PAR_3
	calledNetworkAddress	TS_PAR_4
	connectionIdentifier	’1234567’H
	data	t2 a_constraint
		-- *a_constraint is a formal parameter containing a whole PDU.*
}		

This can be referenced from the Test Case, Test Step or Default behaviour tables, as for example:

N_DATAreq_With_TCON(TCON_Class4_2)

Since the actual parameter is a constraint name, which itself can be parameterized, it is possible to express an arbitrary depth of nesting.

I.2.2.5 Modified constraints – New constraints may be constructed by modifying already defined constraints using the REPLACE mechanism:

ASN.1 PDU Constraint Declaration	
Constraint Name	**:** TCON_Class0_1
PDU Type	**:** T_CONNECT1
Derivation Path	**:** T_CON_Class4_1
Comments	**:**
Constraint Value	
REPLACE t_Class BY 0	

Wildcards can be used as replacements as well:

ASN.1 PDU Constraint Declaration	
Constraint Name	**:** TCON_AnyClass
PDU Type	**:** T_CONNECT1
Derivation Path	**:** T_CON_Class4_1
Comments	**:**
Constraint Value	
REPLACE t_Class BY ?	

To specify fields that shall be omitted, the OMIT mechanism is used. This is only allowed if the field is declared as OPTIONAL:

ASN.1 PDU Constraint Declaration	
Constraint Name	**:** TCON_NoUserData
PDU Type	**:** T_CONNECT1
Derivation Path	**:** TCON_Class4_1.TCON_AnyClass
Comments	**:**
Constraint Value	
OMIT userData	

It is possible to modify ASN.1 parameterized constraints, but note that the parameterized fields themselves cannot be replaced:

ASN.1 PDU Constraint Declaration	
Constraint Name	**:** TCON_2(class:INTEGER)
PDU Type	**:** T_CONNECT1
Derivation Path	**:** TCON_1
Comments	**:**
Constraint Value	
REPLACE userData BY "CPS"	

I.2.3 Further examples of ASN.1 constraints

I.2.3.1 Definition of an FTAM F_INITIALIZEresponse PDU, made in an ASN.1 PDU type definition table:

ASN.1 PDU Type Definition		
ASP Name	**:**	F_INITIALIZEresponse
PCO Type	**:**	
Comments	**:**	
Type Definition		
SEQUENCE {		
state_result		State_Result DEFAULT success,
action_result		Action_Result DEFAULT success,
protocol_version		Protocol_Version DEFAULT { version_1},
implementation_information		Implementation_Information OPTIONAL,
presentation_context_management		[2] IMPLICIT BOOLEAN DEFAULT FALSE,
service_class		Service_Class DEFAULT { transfer_class },
functional_units		Functional_Units,
attribute_groups		Attribute_Groups DEFAULT { },
shared_ASE_information		Shared_ASE_Information OPTIONAL,
ftam_quality_of_service		FTAM_Quality_Of_Service,
contents_type_list		Contents_Type_List OPTIONAL,
diagnostic		Diagnostic OPTIONAL,
checkpoint_window		[8] IMPLICIT INTEGER DEFAULT 1
}		

The fields of the PDU (State_Result, Action_Result, etc.) are declared in ASN.1 Type Definitions.

For example, Functional_Units:

ASN.1 PDU Type Definition		
Type Name	**:**	Functional_Units
Comments	**:**	
Type Definition		
[4] IMPLICIT BITSTRING		
{	read(2),	
	write(3),	
	file_access(4),	
	limited_file_management(5),	
	enhanced_file_management(6),	
	grouping(7),	
	fadu_locking(8),	
	recovery(9),	
	restart_data_transfer(10)	
}		

A base constraint, F_INITrsp_001, on the F-INITIALIZEresponse is declared In the constraints part:

ASN.1 PDU Constraint Definition		
Constraint Name	**:**	F_INITrsp_001
PDU Type	**:**	F_INITIALIZEresponse
Derivation Path	**:**	
Comments	**:**	
Constraint Value		
{		
state_result		State_Result_001,
action_result		Action_Result_001,
protocol_version		Protocol_Version_001,
implementation_information		Implementation_Information_001,
presentation_context_management		FALSE,
service_class		Service_Class_001,
functional_units		Functional_Units_001,
attribute_groups		Attribute_Groups_001,
shared_ASE_information		Shared_ASE_Information_001,
ftam_quality_of_service		FTAM_Quality_Of_Service_001,
contents_type_list		Contents_Type_List_001,
diagnostic		Diagnostic_001,
checkpoint_window		1
}		

A constraint on Functional_Units, Functional_Units_001, is declared in an ASN.1 PDU field constraint declaration:

ASN.1 Type Constraint Declaration		
Constraint Name	**:**	Functional_Units_001
Structured Type	**:**	Functional_Units
Derivation Path	**:**	
Comments	**:**	
Constraint Value		
'001'B – Write only		

A second constraint, F_INITrsp_002 can be built by modifying the base constraint, F_INIT_rsp001:

ASN.1 PDU Constraint Declaration		
Constraint Name	**:**	F_INITrsp_002
Structured Type	**:**	F_INITIALIZEresponse
Derivation Path	**:**	F_INITrsp_001
Comments	**:**	
Constraint Value		
OMIT	implementation_information,	
REPLACE	presentation_context_management	BY TRUE,
REPLACE	functional_units	BY Functional_Units_002
REPLACE	checkpoint_window	BY ?

where Functional_Units_002 is an ASN.1 PDU Constraint Declaration.

I.3 Base and modified constraints

Suppose that we have the following PDU type definition:

PDU Type Definition		
PDU Name : PDU_B **PCO Type** : **Comments** : This is the declaration of the protocol data unit PDU_B		
Field Name	**Field Type**	**Comments**
FIELD1	INTEGER	
FIELD2	HEXSTRING	
FIELD3	BITSTRING	
FIELD4	BOOLEAN	

A base constraint for PDU_B could be:

PDU Constraint Declaration		
Constraint Name : C0 **PDU Type** : PDU_B **Derivation Path** : **Comments** :		
Field Name	**Field Value**	**Comments**
FIELD1	0	
FIELD2	’FF’H	
FIELD3	’00’B	
FIELD4	TRUE	

A modified constraint C1 to the base constraint C0 could be:

PDU Constraint Declaration		
Constraint Name : C1 **PDU Type** : PDU_B **Derivation Path** : C0 **Comments** :		
Field Name	**Field Value**	**Comments**
FIELD1	1	In the base C0 this field value is 0.

We can further build on C1:

PDU Constraint Declaration		
Constraint Name : C2 **PDU Type** : PDU_B **Derivation Path** : C0.C1 **Comments** :		
Field Name	**Field Value**	**Comments**
FIELD2	–	This field is omitted.
FIELD3	?	Any legal value accepted

Reference to a modified constraint in a behaviour tree is made using its name.

I.4 Type definition using macros

PDU type definition with macro symbol:

PDU Type Definition		
PDU Name : T_CONNECT3		
PCO Type :		
Comments : Illustration of TTCN macro mechanism		
Field Name	**Field Type**	**Comments**
<-	T_AddressGroup	
T_Class	INTEGER0to4	Defined as a simple type
UserData	IA5String	

Structured Type Definition		
Type Name : T_AddressGroup		
Comments :		
Element Name	**Type Definition**	**Comments**
Source	BITSTRING [4]	Length is 4 bits
Destination	BITSTRING [4]	Length is 4 bits

PDU Constraint Declaration		
Constraint Name : T_CON_Class4_3		
PDU Type : T_CONNECT3		
Derivation Path :		
Comments :		
Field Name	**Field Value**	**Comments**
<-	GoodAddress	Reference to the structured type constraint declaration
T_Class	4	
UserData	"one, two, three"	

Structured Type Constraint Declaration		
Constraint Name : GoodAddress		
Structured Type : T_AddressGroup		
Derivation Path :		
Comments :		
Element Name	**Element Definition**	**Comments**
Source	’0101’B	
Destination	’1111’B	

I.5 Use of REPEAT

Test Case Dynamic Behaviour					
Test Case Name : RPT_EX2					
Group : TTCN_EXAMPLES/REPEAT_EXAMPLE2/					
Purpose : To illustrate use of REPEAT and parameter passing by textual substitution					
Default :					
Comments :					
No.	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		(FLAG:=FALSE, COUNTER:=0)			
2		!A	A1		
3		REPEAT STEP2 (FLAG, COUNTER) UNTIL [FLAG OR COUNTER=3]			
4		[FLAG]			
5		!D	D1	PASS	
6		[COUNTER=3]			
7		!E	E1	FAIL	
		STEP2 (F:BOOLEAN; NUMBER: INTEGER)			
8		?B (F:=TRUE)	B1		
9		?C (F:=FALSE, NUMBER:=NUMBER+1)	C1		
Detailed Comments: This example shows how repeated execution of STEP2 can be ended either by reception of message B, or reception of three other messages. In the lines following the REPEAT construct, Boolean expressions are used to describe that in the case where B is received, message D is to be sent, and in the case where three other messages are received, E is to be sent. This example also illustrates the effect of parameter passing by textual substitution. This means that F is replaced by FLAG, and NUMBER is replaced by COUNTER, thus making it possible for FLAG and COUNTER to obtain the results of the assignments in STEP2.					

I.6 Test suite operations

Using a Test Suite Operation to set a checksum:

Test Suite Operation Definition
Operation Name : CRC(P:A_PDU)
Result Type : INTEGER
Comments :
Description
Calculate and return the checksum of the PDU P according to the CRC algorithm. NOTE – In a real ATS, this operation would be described in greater detail.

PDU Constraint Declaration		
Constraint Name : CONS1		
PDU Type : A_PDU		
Derivation Path :		
Comments :		
Field Name	**Field Value**	**Comments**
:	:	
Checksum	?	
:	:	
A_PDU.Checksum := CRC(CONS1) in the appropriate SEND event in a behaviour description will set the Checksum in the constraint CONS1.		

I.7 Example of a Test Suite Overview

In the Test Suite Structure table shown below, a hierarchy of the groups and Test Cases in the suite is defined. Within this structure, test selection expressions are identified which govern the selection of Test Groups and the Test Cases for execution. For example, SELEXP_100 is referenced as the controlling expression for Feature X of the protocol. If Feature X is not supported, none of the Test Cases in the suite which are within the Feature X group will be selected.

Test Suite Structure			
Suite Name : TEST_SUITE_A			
Standards Ref : Recommendations xxxx			
PICS Ref : Recommendation aaaa			
PIXIT Ref : Recommendation bbbb			
test notation(s) : DS test method			
Comments : This is an example only.			
Test Group Reference	**Selection Ref**	**Test Group Objective**	**Page Nr**
FEATURE_X	SELEXP_100	Test optional Feature X	50
FEATURE_A/ATTR_A		Test mandatory Attribute A	50
FEATURE_A/ATTR_A/NEGOTIATION	SELEXP_101	Test optional Attribute A negotiation	50
FEATURE_A/ATTR_A/USAGE		Test Attribute A usage	60
FEATURE_A/ATTR_B e		Test mandatory Feature Y	80

To determine whether or not Feature X is supported, SELEXP_100 must be evaluated. This is done by determining whether or not the Test Suite Parameter in SELEXP_100, i.e. TST_FX, is TRUE. If it is, the processing within the group continues. Note that tests for attribute A will be selected (no expression), but that tests for the optional negotiation feature of Attribute A will only be selected if SELEXP_101 is TRUE.

Test Case Index				
Test Group Reference	**Test Case Id**	**Selection Ref**	**Test Group Objective**	**Page No.**
FEATURE_X/ATTR_A/NEGOTIATION	FX_ANEG_1	SELEXP_102	Req. Attr. A, valid neg.	50
	FX_ANEG_2	SELEXP_102	Req. Attr. A, invalid neg.	52
	FX_ANEG_3		Rcv. Attr. A, invalid neg.	54
	FX_ANEG_4		Rcv. Attr. A, invalid neg.	56
FEATURE_X/ATTR_A/USAGE	FX_AUSE_1	SELEXP_103	Use Attr. A (VAL = 0)	60
	FX_AUSE_2		Rcv. Attr. A	62
	FX_AUSE_3		Rcv. Attr. A	64

If Attribute A negotiation is supported, Test Case FX_ANEG_01 through FX_ANEG_04 are candidates for selection. However, Test Cases "01" and "02" will only be chosen if the additional selection expression SELEXP_102 is TRUE. Test Case FX_ANEG_01 will only be selected if the PICS indicates that a value of zero for Attribute A is supported.

The PICS and PIXIT questions used in the test selection expressions are declared as Test Suite Parameters.

Test Suite Parameter Declarations			
Parameter Name	**Type**	**PICS/PIXIT Ref**	**Comments**
TSP_FX	BOOLEAN	PICS question FX1	Q: Feature X supported?
TSP_FXA_N	BOOLEAN	PICS question FX2	Q: Feature X neg supported?
TSP_FXA_NINT	BOOLEAN	PICS question FX3	Q: Does IUT req. neg?
TSP_FXA_MINVAL	INTEGER	PIXIT question FXVAL	Q: Will IUT use VAL = 0?

The test selection expressions are declared as Boolean expressions, as defined in 11.5.

Test Case Selection Expression Definitions		
Expression Name	**Selection Expression**	**Comments**
SELEXP_100	TSP_FX	Feature X supported
SELEXP_101	TSP_FXA_N	Feature X negotiation
SELEXP_102	TSP_FXA_NINIT	Req. Feature X negotiation
SELEXP_103	TSP_FXA_VAL=0	Accept Feature X VAL = 0

I.8 Example of a Test Case in TTCN.MP Form

For the sample Test Case given below:

Test Case Dynamic Behaviour					
Test Case Name	**:** PACKET/P4/PROPER/T_02				
Group	**:** T_7_02				
Purpose	**:** Verify the IUT acknowledges a Clear cause code 05 while in state p4.				
Default	**:**				
Comments	**:**				
No.	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
0		+R1_PREAMBLE(SVC)			
1		+P4D1_PREAMBLE			
2		!CLEAR START TD	CLR_0(LC)		clear cause = 5
3	L1	?CLEARC CANCEL TD	CLRC_0(LC)	(PASS)	
4		+R1_POSTAMBLE			
5		?CLEARC CANCEL TD	CLR_L0(LC)	(PASS)	
6		+R1_POSTAMBLE			
7		?RESTART [RST_ON_ERR] CANCEL TD	STRT_DTEA	(PASS)	
8		!RESTARTC	STRTC		
9		+R1_POSTAMBLE			
10		+DIC_UNEXPECTED			
11		->L1			
12		+RSRT_UNEXPECTED			
13		?TIMEOUT TD		FAIL	
14		?OTHERWISE CANCEL TD		FAIL	

The TTCN.MP that corresponds to this table is:

$BeginTestCase
  $TestCaseId T_7_02
  $TestGroupRef PACKET/P4/PROPER/T_02
  $TestPurpose /* Verify the IUT acknowledges a Clear cause code 05 while in state p4 */
  $DefaultsRef
  $BehaviourDescription
    $BehaviourLine
      $Label
      $Line [0] +R1_PREAMBLE(SVC)
      $Cref
      $Verdict
    $End_BehaviourLine
    $BehaviourLine
      $Label
      $Line [1] +P4D1_PREAMBLE
      $Cref
      $Verdict
    $End_BehaviourLine
    $BehaviourLine
      $Label
      $Line [2] !CLEAR START TD
      $Cref CLR_0(LC)
      $Verdict
      $Comment /* clear cause = 5 */

$End_BehaviourLine
$BehaviourLine
$Label L1
$Line [3] ?CLEARC CANCEL TD
$Cref CLRC_0(LC)
$Verdict (PASS)
$End_BehaviourLine
$BehaviourLine
$Label
$Line [4] +R1_POSTAMBLE
$Cref
$Verdict
$End_BehaviourLine
$BehaviourLine
$Label
$Line [3] ?CLEAR CANCEL TD
$Cref CLR_L0(LC)
$Verdict (PASS)
$End_BehaviourLine
$BehaviourLine
$Label
$Line [4] +R1_POSTAMBLE
$Cref
$Verdict
$End_BehaviourLine
$BehaviourLine
$Label
$Line [3] ?RESTART [RST_ON_ERR] CANCEL TD
$Cref STRT_DTEA
$Verdict (PASS)
$End_BehaviourLine
$BehaviourLine
$Label
$Line [4] !RESTARTC
$Cref STRTC
$Verdict
$End_BehaviourLine
$BehaviourLine
$Label
$Line [5] +R1_POSTAMBLE
$Cref
$Verdict
$End_BehaviourLine
$BehaviourLine
$Label
$Line [3] +D1C_UNEXPECTED
$Cref
$Verdict
$End_BehaviourLine
$BehaviourLine
$Label
$Line [4] -> L1
$Cref
$Verdict
$End_BehaviourLine
$BehaviourLine
$Label
$Line [3] +RSRT_UNEXPECTED
$Cref
$Verdict
$End_BehaviourLine
$BehaviourLine
$Label
$Line [3] ?TIMEOUT TD
$Cref
$Verdict FAIL

$End_BehaviourLine

$BehaviourLine

$Label

$Line [3] ?OTHERWISE CANCEL TD

$Cref

$Verdict FAIL

$End_BehaviourLine

$End_BehaviourDescription

$End_TestCase

The layout shown here is only intended to aid readability.

I.9 Use of Component Reference for Field Value Assignment in Constraints

When a number of field values in a received PDU must be assigned to the fields in several subsequent send PDUs, the Dynamic Behaviour table can become cluttered with lengthy assignment statements using the dot notation.

TTCN allows PDU field value assignments in the constraint tables using component reference associated with a formal parameter. Received ASPs or PDUs in the Behaviour table may be assigned to a variable and subsequently passed as an actual parameter in the constraints reference to a formal parameter in the constraint table. The constraint table then specifies the required field assignments using the formal parameter and its components. The following tables illustrate these principles.

Figure I.1 illustrates possible field assignments in the behaviour specification without the use of component reference.

Test Case Dynamic Behaviour					
Test Case Name : TTCN_EXAMPLES/STYLE1 **Group** : ST_EX1 **Purpose** : To illustrate the use of component references in the behaviour description **Default** :					
No.	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1 2		?InASP(v:=InASP.userdata) !OutASP (OutASP.userdata.OutPDU.FieldA:=v.Field2; OutASP.userdata.OutPDU.FieldC:=v.Field3)	Cin1 Cout1		

Figure I.1/X.292 – Lengthy assignment statements clutter the behaviour description

Figure I.2 illustrates the simplification of the behaviour specification resulting from the use of component reference in constraints.

For simplicity, the definitions of all required ASP and PDU types have been omitted.

The ASP types InASP and OutASP consist of the single parameter field userdata, which is of the type InPDU and OutPDU respectively. InPDU contains the three fields Field1, Field2 and Field3, which all are of the type IA5String.

OutPDU contains the three fields FieldA, FieldB and FieldC, which also are of the type IA5String.

v has to be declared as a Test Case Variable of a PDU type.

Test Case Dynamic Behaviour					
Test Case Name : TTCN_EXAMPLES/STYLE1					
Reference : ST_EX1					
Purpose : To illustrate the use of component references in the behaviour description					
Default :					
No.	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		?InASP(v:=InASP.userdata)	Cin1		
2		!OutASP	Cout2(v)		

Figure I.2/X.292 – Lengthy assignment statements are removed from the behaviour description

The following tables give the required ASP and PDU constraint declarations:

ASP Constraint Declaration		
Constraint Name : Cout1		
ASP Type : OutASP		
Derivation Path :		
Comments :		
Parameter Name	**Parameter Value**	**Comments**
Userdata	CoutPDU1	

ASP Constraint Declaration		
Constraint Name : Cout2(p:**PDU**)		
ASP Type : OutASP		
Derivation Path :		
Comments :		
Parameter Name	**Parameter Value**	**Comments**
Userdata	CoutPDU2(p)	

ASP Constraint Declaration		
Constraint Name : Cin1		
ASP Type : InASP		
Derivation Path :		
Comments :		
Parameter Name	**Parameter Value**	**Comments**
Userdata	CinPDU	

PDU Constraint Declaration		
Constraint Name : CoutPDU1		
PDU Type : OutPDU		
Derivation Path :		
Comments :		
Field Name	**Field Value**	**Comments**
FieldA	'A'	
FieldB	'B'	
FieldC	'C'	

PDU Constraint Declaration		
Constraint Name : CoutPDU2(p:PDU)		
PDU Type : OutPDU		
Derivation Path :		
Comments :		
Field Name	**Field Value**	**Comments**
FieldA	p.Field2	
FieldB	'B'	
FieldC	p.Field3	

PDU Constraint Declaration		
Constraint Name : CinPDU		
PDU Type : InPDU		
Derivation Path :		
Comments :		
Field Name	**Field Value**	**Comments**
Field1	*	
Field2	*	
Field3	*	

I.10 Multi-Party Testing

Figure I.3 illustrates a test component configuration for a typical multi-party testing context. Only a single upper tester is shown, since communication among multiple upper testers and/or Upper Tester Control Function (UTCF) is only applicable to contexts that exclusively use the local test method.

In the example shown in Figure I.3, for simplicity, each lower tester is specified by a single PTC and the LTCF is specified by the MTC. Another PTC is used to specify the upper tester. Coordination points are used between the lower tester PTCs and the MTC.

This is a straightforward use of concurrency to meet multi-party requirements, but it should not be taken to imply that there has to be a one-to-one relationship between lower testers and PTCs, or between the LTCF and the MTC, or between the upper tester and a PTC.

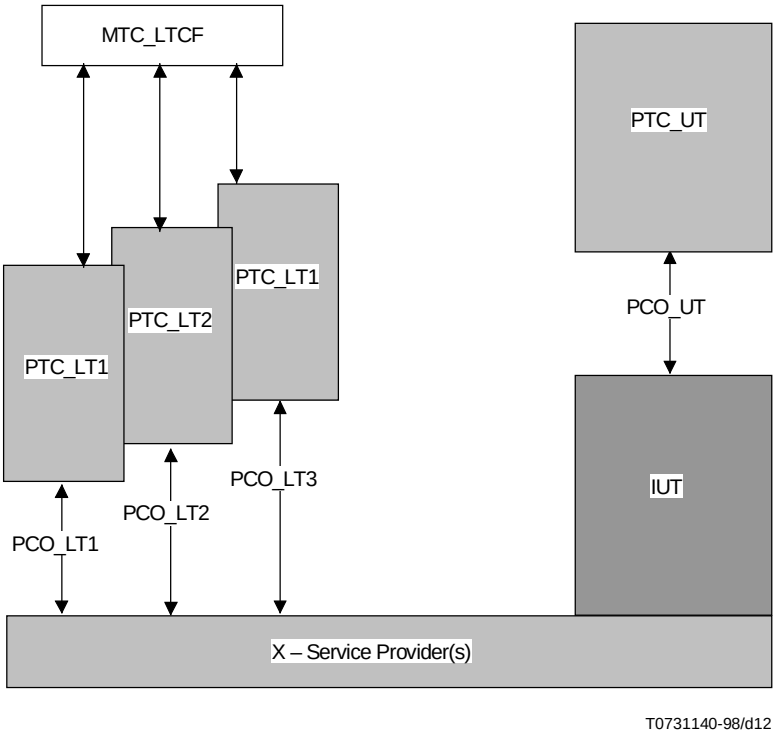


Figure I.3/X.292 – Example Test Component Configuration for Multi-Party Testing with a Single Upper tester

I.11 Multiplexing/Demultiplexing

There are two ways of using concurrent TTCN in test cases using multiplexing/demultiplexing. These are illustrated in Figure I.4. The first, shown in Figure I.4 a), specifies the multiplexing and demultiplexing explicitly within test component MTC1, with PTC1 and PTC2 each handling the behaviour on one of the two multiplexed connections. This provides for maximum flexibility in the way that the multiplexing and demultiplexing behaviour is specified, including possibilities of invalid behaviour. However, the disadvantage of this approach is that the relatively complex multiplexer/demultiplexer has to be specified even if the test purpose concerns only the behaviour on each of the two connections. The alternative approach is to use a separate PCO for each separate stream of events and a test suite parameter (MuxValue) associated with each of these PCOs that are to be multiplexed and demultiplexed within the underlying service provider, rather than within the Lower Tester. This allows the configuration shown in Figure I.4 b) to be used. Since the multiplexing/demultiplexing is performed within the service provider, there are two PCOs in this configuration, corresponding to the two CPs in the other configuration, but they are given a common MuxValue, MuxA, to indicate that within the service provider they are to be multiplexed. To keep things simple, one of the two test components is made the MTC, although a separate MTC not connected to a PCO could be used instead if preferred.

I.12 Splitting and Recombining

In order to specify test cases involving splitting and recombining, there is no alternative to specifying explicitly the splitting and recombining behaviour in the test case. Concurrency can be used to separate the splitting and recombining behaviour into one test component, MTC1 in Figure I.5, from the protocol behaviour that lies above this function by using a second test component, PTC1 in Figure I.5.

I.13 Multi-Protocol Test Cases

Multi-protocol test cases, including those using the embedded variants of the test methods, can use concurrent TTCN in order to separate the behaviour associated with each protocol into a different test component, as illustrated in Figure I.6, which shows an example configuration for testing Session embedded under FTAM.

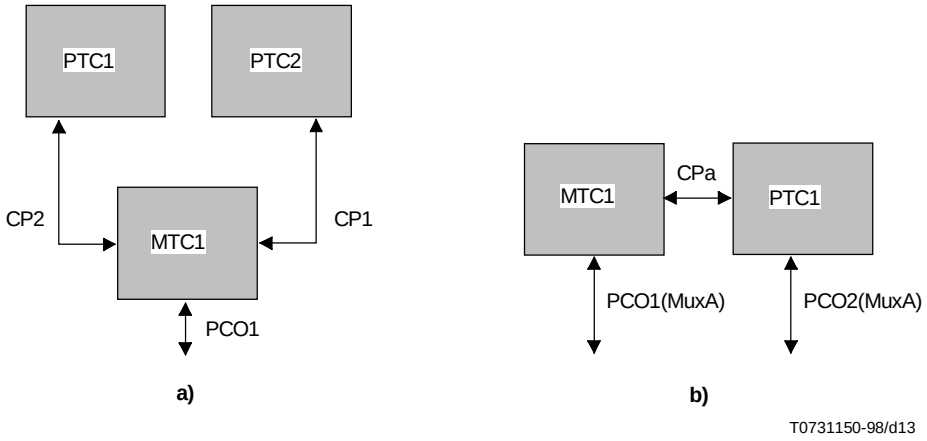


Figure I.4/X.292 – Possible Configurations for Multiplexing/Demultiplexing Tests Cases

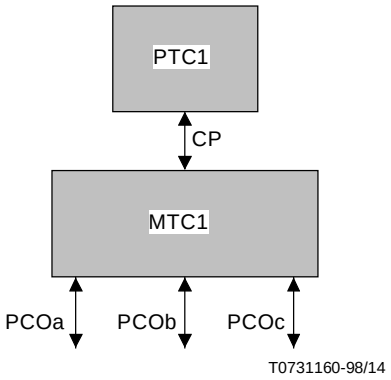


Figure I.5/X.292 – Possible Configuration for Splitting/Recombining Test Cases

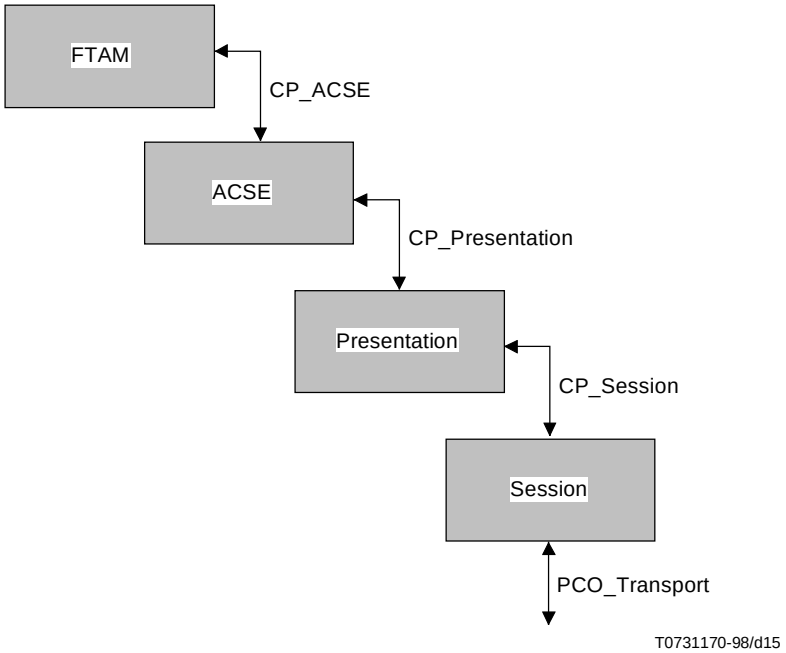
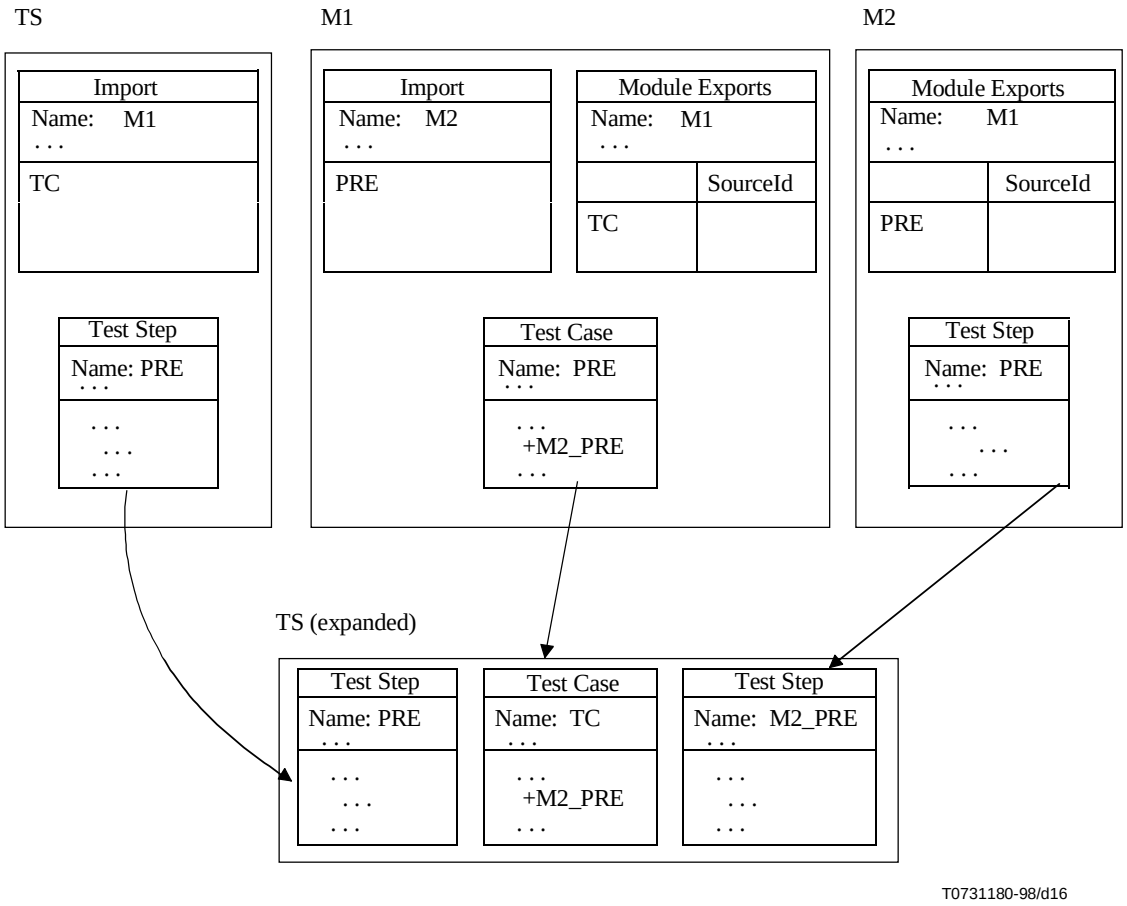


Figure I.6/X.292 – Possible Configuration for Multi-Protocol Testing – Session embedded under FTAM

I.14 Example of Modular TTCN



The test step PRE (which is defined in the module M2) is implicitly imported from M1 in TS.

I.15 Example of CREATE and DONE

NOTE – An extra example will be added in the published second edition to clarify the use of CREATE and DONE, especially with regard to the implicit passing of preliminary results and verdicts, giving an explanation of the semantics by showing the same test case specified using explicit passing of variables using CMs and CPs.

Test Suite Overview:

Test Suite Structure		
Suite Name : Done		
Standards Ref :		
PICS Ref :		
PIXIT Ref :		
Test Method(s) :		
Comments : This test suite demonstrates a possible interpretation of CREATE and DONE statements as well as verdict and variable passing. The interpretation is in terms of communication using CMs and existing CPs.		
Test Group Reference	**Selection Ref**	**Test Group Objective**
Detailed Comments:		

Test Case Index				
Test Group Reference	**Test Case Id**	**Selection Ref**	**Description**	**Page No.**
	TC1 TC1_EXPANDED			241 242
Detailed Comments:				

Declarations Part:

ASN.1 Type Definition	
Type Name	**:** TestStop
Comment	**:**
Type Definition	
IA5String	
Detailed Comments:	

ASN.1 Type Definition	
Type Name	**:** VariableList
Comments	**:** Should really be some representation of variables and their values.
Type Definition	
NULL	
Detailed Comments:	

Test Case Variable Declarations			
Variable Name	**Type**	**Value**	**Comments**
Verdict	R_Type	fail	
Detailed Comments:			

PCO Declarations			
PCO Name	**PCO Type**	**Role**	**Comments**
PCO1	XSAP	LT	
Detailed Comments:			

CP Declarations	
CP Name	**Comments**
CP1	
Detailed Comments:	

Test Component Declarations				
Component Name	**Component Role**	**No. of PCOs**	**No. of CPs**	**Comments**
PTC1	PTC	1	1	
Master	MTC	0	1	
Detailed Comments:				

Test Component Configuration Declaration			
Configuration Name : Conf1			
Comments :			
Components Used	**PCOs Used**	**CPs Used**	**Comments**
Master		CP1	
PTC1	PCO1	CP1	
Detailed Comments:			

ASP Type Definition		
ASP Name : ASP1		
PCO Type : *XSAP*		
Comments :		
Parameter Name	**Parameter Type**	**Comments**
Detailed Comments:		

ASP Type Definition		
ASP Name : ASP2		
PCO Type : *XSAP*		
Comments :		
Parameter Name	**Parameter Type**	**Comments**
Detailed Comments:		

CM Type Definition		
CM Name : CM1		
Comments :		
Parameter Name	**Parameter Type**	**Comments**
Detailed Comments:		

CM Type Definition		
CM Name : CREATE_CM **Comments** :		
Parameter Name	**Parameter Type**	**Comments**
StepName Variables	TestStep VariableList	
Detailed Comments:		

CM Type Definition		
CM Name : DONE_CM **Comments** :		
Parameter Name	**Parameter Type**	**Comments**
Verdict Variables	R_Type VariableList	
Detailed Comments:		

Constraints Part:

CM Constraint Declaration		
Constraint Name : Create(Step:TestStep) **CM Type** : CREATE_CM **Derivation Path** : **Comments** :		
Parameter Name	**Parameter Value**	**Comments**
StepName Variables	Step NULL	
Detailed Comments:		

CM Constraint Declaration		
Constraint Name : Done **CM Type** : DONE_CM **Derivation Path** : **Comments** :		
Parameter Name	**Parameter Type**	**Comments**
Verdict Variables	? NULL	
Detailed Comments:		

Dynamic Part:

Test Case Dynamic Behaviour					
Test Case Name : TC1					
Group :					
Purpose :					
Configuration :					
Default :					
Comments : Normal style concurrent TTCN					
No.	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		CREATE(PTC1: TS1)			
2		CP1!CM1			
3		?DONE(PTC1)		R	
		TS1			
4		CP1?CM1			
5		PCO1!ASP1			
6		PCO1?ASP2		(P)	
Detailed Comments:					

Test Case Dynamic Behaviour					
Test Case Name : TC1_EXPANDED					
Group :					
Purpose :					
Configuration :					
Default :					
Comments : This is the same Test Case as TC_1 except the CREATE and DONE statements are explicitly implemented through special CMs being transmitted over a CP. The only limitation in TTCN is that constraints may not accept the name of a Test Step as the parameter. This is emulated in this example as by simple string_matching. An enum constructed from all the Test Step names would be more elegant. Every PTC would need a PTC_DISPATCHER as the one in this example.					
No.	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		CREATE_CM	Create("TS1_EXPANDED")		Corresponds to CREATE (PTC1:TS1)
2		CP1!CM1			
3		CP1?DONE_CM(Verdict := DONE_CM.Verdict)	Done		
4		[Verdict=pass]		P	
5		[Verdict=fail]		F	
6		[Verdict=inconc]		I	
		PTC1_DISPATCHER			
		CP1?CREATE_CM	Create("TS1_EXPANDED")		
		+TS1			
		CP1!DONE_CM(DONE_CM.Verdict:=R)	Done		
		TS1			
		CP1?CM1			
		PCO1!ASP1			
		PCO1?ASP2		(P)	
Detailed Comments:					

Appendix II

Style guide

II.1 Introduction

This appendix presents some recommended style rules that can be employed when using TTCN. The aim is to provide a basic consistency between the TTCN styles used by different test suite specifiers.

II.2 Test case structure

In order to have a better analysis of test results and to identify easily whether or not the test purpose is achieved, the consideration of the following points on structuring Test Cases is suggested:

a) the test suite specifier should clearly identify the preamble and postamble subtrees;

b) the postamble and the preamble should be specified through a single test tree attachment (local to the Test Case or from the Test Step Library) in the Test Case main behaviour tree. Such test trees may attach subsequent subtrees;

c) once the preamble and postamble(s) subtrees are identified within a Test Case main behaviour tree, the remaining events in the Test Case main behaviour tree may be considered to be related to the test body (i.e. events related to the test purpose).

Using this mechanism, the boundaries between preamble, test body and postamble within a Test Case can be easily identified. Labels may be used to indicate the start and end of the test body in the conformance log.

Test Case Dynamic Behaviour

Test Case Name : TTCN_EXAMPLES/STYLE1
Group : ST_EX1
Purpose : To illustrate identification of preambles and postambles.
Configuration :
Default :
Comments :

No.	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		+Preamble			
2		!A	A1		Related to purpose
3	Body	?B	B1		Related to purpose
4	CinBody	?C	C1	(PASS)	Related to purpose
5		+postamble_1			
6	DinBody	?D	D1	(PASS)	Related to purpose
7		+postamble_2			
8		?E	E1	INCONC	Related to purpose
9		?OTHERWISE		FAIL	

Figure II.1/X.292 – Identification of preambles and postambles

Since final verdicts cause termination of Test Case execution, a test suite specifier cannot assign a final verdict in the body if it is necessary to enter the postamble. Still, it is desirable to give a verdict at the point in the Test Case where the test purpose is achieved and not hide verdicts in postambles. It is therefore recommended to state preliminary results in the verdict column if a test purpose is achieved but a postamble should still be executed. In the definition of the postamble, a test suite specifier may use the result variable R as a verdict assigned at the leaves of the behaviour tree, to indicate that if no errors were encountered in the postamble, the verdict is determined in the test body.

II.3 Use of TTCN with different abstract test methods

II.3.1 Introduction

This subclause ties the TTCN with the abstract test methods defined in Recommendation X.291. It gives the TTCN syntax used to express the occurrence of events at PCOs, and constraint references for the various abstract test methods.

It is assumed that the ASP type definitions define the type of the UserData parameter as PDU. It is therefore possible to use chaining of constraints (i.e. to refer to a constraint for an ASP that contains a PDU in the UserData parameter) as a reference to an ASP constraint that has a PDU constraint as an actual parameter.

II.3.2 TTCN and the LS test method

Possible TTCN events:

Behaviour Description	*Constraints Reference*
LT! N_ASP	N_ASPconstraint (N_PDUconstraint)
LT? N_ASP	N_ASPconstraint (N_PDUconstraint)
UT! T_ASP	T_ASPconstraint
UT? T_ASP	T_ASPconstraint

II.3.3 TTCN and the DS test method

Possible TTCN events:

Behaviour Description	*Constraints Reference*
LT! N_ASP	N_ASPconstraint (T_PDUconstraint)
LT? N_ASP	N_ASPconstraint (T_PDUconstraint)
UT! T_ASP	T_ASPconstraint
UT? T_ASP	T_ASPconstraint

II.3.4 TTCN and the CS test method

Possible TTCN events:

Behaviour Description	*Constraints Reference*
LT! N_ASP	N_ASPconstraint (T_PDUconstraint)
LT? N_ASP	N_ASPconstraint (T_PDUconstraint)

Exchanging TM_PDUs between the LT and TM protocol implementation in the IUT, via the connection that is used for testing. Note that in this case the PDU definition shall have declared its UserData field as of type PDU.

LT! N_ASP	N_ASPconstraint (T_PDUconstraint (TM_PDUconstraint))
LT? N_ASP	N_ASPconstraint (T_PDUconstraint (TM_PDUconstraint))

II.3.5 TTCN and the RS test method

Possible TTCN events:

Behaviour Description	*Constraints Reference*
LT! N_ASP	N_ASPconstraint (T_PDUconstraint)
LT? N_ASP	N_ASPconstraint (T_PDUconstraint)

Since there is no UT or TMP, the IMPLICIT SEND is used to describe send events at the side of the IUT connection.

<IUT! N_ASP>	N_ASPconstraint (T_PDUconstraint)
<IUT! T_PDU>	T_PDUconstraint

II.4 Use of Defaults

As a matter of style, a test suite specifier should avoid situations where the attempt of an alternative of a Default behaviour is the normal specification of the *expected* behaviour of the IUT. It would be the case for instance if a Test Step represents the behaviour of the LT or UT and the IUT, when valid test events are sent, and if the responses of the IUT to invalid or inopportune test events sent by the LT or UT were specified in Defaults implicitly attached to that Test Step when called by other Test Cases. Such Defaults would have to bear Pass verdicts.

This is not a recommended practice when the attachment of a Default tree is left unspecified and carries a degree of uncertainty. Explicitly attached trees or the main tree should be used instead.

II.5 Limiting the execution time of a Test Case

In previous versions of TTCN, an ELAPSE statement was defined, allowing the test case specifier to limit the abnormal duration of a Test Case, if for instance a snapshot processing never ends, or if an uncontrolled recursion of tree attachment occurs.

The ELAPSE statement is no longer part of TTCN, as the problem it was intended to solve is considered to be outside the scope of the test suite specification.

To limit the execution time of a Test Case, it is now recommended that the test realizers implement local mechanisms in the means of testing. Explicit timers can be used together with the TIMEOUT event whenever a limit needs to be placed on waiting for an event to occur.

II.6 Structured Types

a) In pre-DIS versions of TTCN, generic fields and generic values were defined as features allowing either to group several fields or values in a constraint table, and/or to re-use such a group in several constraint tables of similar contents.

b) In this version, the grouping of ASP parameters and PDU (ex-data types) fields is introduced first in the declarations part, for the sake of completeness of that part, and consistency with the use of ASN.1 in TTCN. Refer to 11.2.3.3 for a definition of the Structured Type definition tables. Once a Structured Type is declared, it can be used by one or more ASP type or PDU type definitions. The ASP and PDU definition table can therefore be "flat" (no group, or a group introduced by a macro call), or structured (by means of structure specifications for named ASP parameters or PDU fields).

c) In the constraint part, structure elements must be assigned values in Structured Type constraint tables. The names of these constraints can be used in the base ASP or PDU constraint tables as values.

   The ASP and PDU constraint tables can therefore also be:

   – flat, i.e. assigning values to all parameters or fields individually, and only referring to the structure constraint tables by macro call; or

   – structured, i.e. replacing values of declared groups of parameters or fields by names of group constraints.

d) If the declared ASP or PDU is structured by use of some ASP parameters or PDU fields being specified by reference to structure elements, then the constraints have to have the same structure.

Whichever form is used, ASP/PDU constraints can also be:

– modified; and

– parameterized, by means of a parameter to be bound to a field/parameter value or to a Structured Type constraint.

e) The Structured Type constraint tables replace the generic field tables of previous versions of TTCN.

f) The concept of generic values is deleted.

g) Examples are given in Appendix I.

II.7 Abbreviations

In previous versions of TTCN, it was allowed to declare, in a specific table, abbreviations to be used in the behaviour columns of the Test Cases and Test Steps. This facility proved to be confusing and has been restricted so that only the names of ASPs and PDUs, when used in event lines, can be abbreviated. This facility is now called Alias.

II.8 Test descriptions

Informal behaviour descriptions, giving more detail than the test purposes, but less detail than the TTCN specification of the Test Cases may, if desired, be included in a standardized ATS.

Such test descriptions may use text, time sequence diagrams or any other notation and be located in the comments field of tables, an informative annex or both.

The TTCN specifications of the Test Cases always take precedence over such informal test descriptions.

II.9 Assignments on SEND events

TTCN allows for overwriting constraint values prior to a SEND event in an assignment statement on the event line. This means that first the data to be sent is constructed from the constraint definition and then the assignments are executed.

This feature may cause the test suite reader confusion concerning the value to be sent, and therefore it should be used with caution. In particular, it is considered bad style to use the same constraint for both sending and receiving.

II.10 Multi-service PCOs

Where a PCO covers more than one SAP, the precise specification of such a PCO is given by the set of ASPs and PDUs that can occur.

EXAMPLE II.1 – An FTAM PCO:

PCO Declarations			
PCO Name	**PCO Type**	**Role**	**Comments**
L	A_P_SAPs	LT	PCO through which we can observe all ACSE ASPs and all Presentation ASPs except P-CONNECT, P-RELEASE and P_ABORT.

The PCO "L" is of type A_P_SAPs which is able to observe all ACSE and Presentation ASPs, excluding P-CONNECT, P-RELEASE and P-ABORT. The type column shows which SAPs belong to the set to be observed by the PCO, "A" and "P", each SAP separated by underscore ("___"). The comments column describes exactly what can be seen by the PCO.

This method is extensible to many SAPs, each of which would be separated by an underscore.

Appendix III

Index

III.1 Introduction

This appendix presents an alphabetical index of terms and acronyms used in this Recommendation. For each term or acronym, the index gives a set of references in terms of clause, figure and table numbers, either in the main body, or in the annexes and appendixes of this Recommendation. The significance of each reference is indicated as follows:

a) definitions of the terms and acronyms are in **bold**;

b) major uses of the term or acronyms are in *italics*;

c) other uses are in normal font.

III.2 The Index

A

ABSENT: *A.4.2.5*

Abstract Service Primitive: *1*, *4.1*

Abstract Syntax Notation One: *4.3*

ABSTRACT SYNTAX: *A.4.2.5*

Abstract test case: *6*, *8.2*, *11.13.2*, *15.17.1*

Abstract test suite: *1*, *2*, *4.1*, *8.2*

Abstract testing methodology: *1*

Access to behaviour description: 15.13.2

ACTIVATE procedure: **B.5.19.1**

ACTIVATE: *15.4.1*, *15.14*, **15.18.4**, *15.18.4*, 15.18.6, *15.18.6*, *A.4.2.4*, *B.5.5.4*, *B.5.5.5*, *B.5.18.2*, *B.5.19.2*

Actual parameters: *15.13.5*, *15.16.2*

ActualParList: *B.5.5.3*

Alias definition: *11.1*, *A.3.3.13.14*

ALL: *A.4.2.5*

Ancestor node: *15.14*

AND: *A.4.2.4*

AnyOne: **12.6.5.1**, *12.6.6.1*

AnyOrNone: **12.6.5.2**, *12.6.5.3*, *12.6.6.1*

AnyOrOmit: **12.6.4.4**, *12.6.6.1*

AnyValue: **12.6.4.3**, *12.6.5.1*, *12.6.6.1*

APPEND_DEFAULTS: **B.5.5.4**

APPEND_TO_LEVEL: **B.5.25**

Applicable encoding rules: **3.6.1**

APPLICATION: *A.4.2.5*

Arithmetic operators: 11.3.2.2

Array references: 15.10.2.3

ASN.1 ASP constraints: *14.2*, 14.3, *A.4.2.15*

ASN.1 ASP type definition: 11.14

ASN.1 CM constraints: 14.9, *A.4.2.15*

ASN.1 CM type: *11.17.3*

ASN.1 comments: 11.2.3.4, *11.15.4*, *14.1*

ASN.1 compact constraints: E.2.5

ASN.1 constraint declaration: 12.6.6.1, 14, A.3.3.22, E.2.5, I.2

ASN.1 constraint: 12.6.6.2

ASN.1 constraints: 12.1, 12.6.5.2, 14.1

ASN.1 dash symbol: 14.1

ASN.1 defined data objects: 15.10.2

ASN.1 encoding rules: 11.15.1

ASN.1 identifier: 3.6.48

ASN.1 module: 11.2.3.5, 11.14.5

ASN.1 PDU constraint declaration: 14.4

ASN.1 PDU constraints: 14.2, A.4.2.15

ASN.1 PDU type definition: 11.15

ASN.1 type constraints: 7.3.4, 11.16.4, 14.2, A.4.2.15

ASN.1 type definition: 11.2.3.4, 11.2.3.5, 11.18.2, 14.5, 14.8, A.4.2.1, A.4.2.6

ASN.1 type: 11.2.3.4, 11.2.3.5, 11.8.1, 11.8.3, 11.14.2, 11.14.5, 11.15.4, 12.6.2

ASN.1: 1, 2, 4.3, 8.1, 9.5, 11.2.2, 11.2.3.4, 11.6, 11.7, 11.14.3, 11.14.4, 11.14.5, 11.15.4, 11.15.5, 11.17.3, 12.2, 12.6.1, 12.6.4.2, 15.10.2, A.4.2.1, A.4.2.5, E.2.1, II.6

ASP constraint compact proforma: E.2.2

ASP constraint declaration: 3.6.62, 13.3, d), A.5.1, E.2.5

ASP constraints: 7.3.4

ASP identifier: 11.21

ASP parameter: 3.6.66, 11.2.1, 11.14.2, 12.5, 12.6.2, 12.6.3, 12.6.4.1, 12.6.4.2, 12.6.4.3, 12.6.4.4, 12.6.4.5, 12.6.4.7, 12.6.4.8, 12.6.5.1, 12.6.5.3, 12.6.6.2

ASP specified by reference: 11.14.5

ASP type definition: 3.6.3, 3.6.68, 11.1, 11.2.2, 11.14, 11.19, 11.20, A.3.3.19, A.3.3.22, II.3.1

ASP type: 11.3.4.2, 14.3, 15.7.2

ASP: 3.6.9, 3.6.13, 3.6.25, 3.6.38, 3.6.44, 3.6.57, 3.6.60, 3.6.68, 4.1, 8.1, 9.5, 11.2.1, 11.2.2, 11.2.3.3, 11.3.4.1, 11.3.4.2, 11.6, 11.7, 11.10, 11.14, 11.14.2, 11.14.3, 11.14.4, 11.14.5, 11.15, 11.15.1, 11.15.5, 11.16.4, 11.19, 11.20, 11.21, 12.1, 12.4, 12.6.1, 12.6.3, 13.2, 13.6, 14.5, 14.6, 14.8, 15.2.1.3, 15.9, 15.9.5.3, 15.9.6, 15.10.1, 15.10.2.2, 15.10.2.3, 15.10.3, 15.10.6, 15.16.1, A.4.2.7, A.4.2.8, B.5.2.3, B.5.8.2, B.5.9.2, B.5.10.2, B.5.11.2, B.5.12.2, B.5.13.2, B.5.16.2, E.2.1, II.6, II.7, II.10

ASPs specified in ASN.1: 11.14.4

Assignment rules: 15.10.4.2

Assignment: 11.3.4.3, 11.3.4.6, 11.8.2, 11.8.4, 15.6, 15.8, 15.9.3, 15.9.4, 15.10.1, 15.10.4, 15.10.5, 15.10.6, 15.11, 15.16.3, 15.17.2, B.5.16, II.9

ATS: 3.6.74, 4.1, 6, 10.1, 10.2, 10.3, 10.4, 10.5, 11.1, 11.3.4.1, b), 11.9, 11.14.4, 11.16.1, 12.1, A.1, A.5.1

Attach construct: 3.6.2, 15.2.3, 15.8, 15.17.1, B.5.5.4, B.5.5.5, E.3.1

ATTACH: 15.9.10.1, 15.13.1, 15.13.4.1, E.3.1

Attached tree: 15.13.3

Attachment construct: B.5.1

Attribute: 11.15.2, 11.18.1, 13.4

Attributes of values: 12.6.6

AUTOMATIC: A.4.2.5

B

Backus-Naur Form: 4.3

Base constraint: 3.6.3, 3.6.24, 3.6.44, 13.6, 13.7, A.3.3.19, A.3.3.22, E.2.3, I.3

Base type: 3.6.4, 11.18.2

BEGIN: 11.3.4.4, A.4.2.5

Behaviour description: 3.6.40, 3.6.55, 3.6.78, 3.6.90, 11.10, 11.21, 12.1, 12.3, 15.2.1, 15.2.1.3, 15.2.5, 15.5, 15.13.2, 15.15, A.4.2.10, A.5.1, A.5.2, E.3.1, II.3, II.8

Behaviour line: 3.6.5, 3.6.14, 3.6.25, 15.2.5, B.5.1

Behaviour tree: 3.6.6, 3.6.8, 3.6.42, 3.6.49, 3.6.59, 3.6.83, 3.6.84, 3.6.85, 3.6.87, 15.2.1.3, 15.2.2, 15.4.1, 15.5, 15.9.5, 15.11, 15.13.3, 15.13.4.1, 15.14, 15.16.2, 15.17, 15.18.1, B.5.1, B.5.5.4, B.5.5.5, II.2

BehaviourLine: B.5.2.5

BER: 11.15.2, 11.15.4, 11.15.5, 11.16.4, 13.4, 14.4

Binding of variables: 11.8.4

Bit reference: 15.10.2.4

BIT: A.4.2.5

BIT_TO_INT: 11.3.3.2.1, 11.3.3.2.3, A.4.2.4

BITSTRING: 11.2.2, 11.18, 15.10.2.4, 15.10.4.2, A.4.2.4

Blank entry: 3.6.7

BMPString: A.4.2.5

BNF grammar for TTCN: 7.2

BNF: 4.3, 7.2, A.3

Boolean expression: 15.10.1

Boolean operators: 11.3.2.4

BOOLEAN: 11.2.2, 11.3.3.3.1, 11.3.3.3.2, b), 15.10.2.4, A.4.2.4, A.4.2.5, B.5.15

Bound variable: 11.8.2, 15.16.2, 15.18.2

Bounded free text: 7.4

BUILD_SEND_OBJECT: B.5.8.1

BY: A.4.2.4

C

Calling tree: 3.6.2, 3.6.8, 3.6.42, 15.13.3, 15.17.3, 15.18.5

CANCEL operation: 15.12.3

CANCEL: 15.12.1, 15.12.3, A.4.2.4, B.5.14.2, B.5.17

CANCEL_TIMER: B.5.17.1

CASE OF ELSE: 11.3.4.9

CASE: 11.3.4.9, A.4.2.4

Chaining of constraints: 12.4, 15.10.2.2, 15.10.3, I.1.1.3, I.1.2.3, I.2.2.3, II.3.1

CHARACTER: A.4.2.5

Characterstring type: 11.3.3.3.4, 11.18.1

CharacterString: 11.2.2, 12.6.5.1, 12.6.5.2, 15.10.4.2

CHOICE: 11.3.3.3.2, 12.4, 14.5, 14.8, 15.10.2.2, 15.10.2.3, A.4.2.5

CLASS: A.4.2.5

CM constraint declarations: 13.8

CM parameters: 11.17.1

CM type: 11.17.2, 11.17.3

CM: 3.6.16, 4.3, 8.1, 11.3.4.2, 11.6, 11.7, 11.11, 11.17.1, 11.17.2, 12.1, 13.6, 13.8, 14.9, 15.9.2, 15.9.3, 15.9.4, 15.9.5.3, 15.9.5.4, 15.9.8, *15.10.2.2*, *15.10.2.3*, *15.10.3*, *15.10.6*, *15.16.1*, *15.17.5*, *15.18.8*, *A.4.2.7*, *A.4.2.8*, *B.5.2.3*, *B.5.8.2*, *B.5.9.2*, *B.5.10.2*, *B.5.11.2*, *B.5.12.2*, *B.5.16.2*

CMs and defaults: 15.18.8

Collective comment: 7.3.3, *11.2.3.2*

Compact constraint table: *3.6.7*, **3.6.9**, *13.1*, *E.1*, E.2

Compact proformas: Annex E

Compact test case table: **3.6.10**, *E.1*, E.3

Complement matching operation: 12.6.4.1

COMPLEMENT: *A.4.2.4*

Complement: **12.6.4.1**, *12.6.6.1*

Complex CMs: 11.17.1

Compliance: 6, *15.17.3*

Component of data object: 15.10.2.2, 15.10.2.3

COMPONENT: *A.4.2.5*

Component: *3.6.72*

Concurrent test case behaviour: 15.2.4

Concurrent test case: *3.6.11*, *3.6.72*

Concurrent TTCN: *3.6.11*, **3.6.12**, *3.6.47*

Conflict between TTCN forms: 5

Conformance log: *15.17*, *B.3*, *B.5.20.2*, *B.5.23.2*, B.5.24, *B.5.24.2*, *II.2*

Conformance test suite: *1*

CONSTRAINED: *A.4.2.5*

Constraint declarations: *3.6.25*

Constraints for RECEIVE: 12.6

Constraints part: **3.6.13**, 9.5, 12, *15.2.1.3*, *15.16.1*, *A.3.3.36.2*

Constraints reference: *3.6.5*, **3.6.14**, *3.6.25*, *12.2*, *12.3*, *15.2.1.3*, 15.16, 15.16.1, *B.1*, *B.5.8.2*, *B.5.9.2*, *B.5.10.2*, *B.5.11.2*, *B.5.12.2*, *B.5.13.2*, *II.3*

Construct: *3.6.61*, *3.6.90*, *15.2.1.3*, *15.8*, *15.9.5*, *15.17.1*, *15.18.1*, B.5.18

CONSTRUCT_TYPE_OF: **B.5.26**

Coordinated test method: II.3.4

Coordination message declarations: 8.1

Coordination message: **3.6.15**, *4.3*, *8.1*

Coordination point declarations: 8.1

Coordination point model: 11.11

Coordination point: *3.6.15*, **3.6.16**, *4.3*

CP: *3.6.72*, *3.6.73*, **4.3**, *8.2*, *11.11*, *11.13.1.1*, *11.13.1.3*, *11.13.2*, *15.2.4*, *15.9.5.3*, *15.9.8*, *15.9.10.1*, *15.10.6*, *A.4.2.4*, *A.4.2.13*, *B.1*, *B.5.4.2*, *I.11*

CREATE and defaults: 15.18.7

Create construct: **15.9.10.1**

CREATE procedure: **B.5.20.1**

CREATE: *8.2*, *11.13.1.1*, *11.13.1.2*, *15.9.10*, *15.9.10.1*, *15.9.10.2*, *15.18.7*, *A.4.2.4*, *B.5.18.2*, I.15

CurrentLevel: *B.5.2.3*

D

Data object: *12.2*, *15.10.1*

Declaration by reference: 11.7

Declarations part: **3.6.17**, 9.5, *11.1*, *15.9.1*, *A.3.3.36.2*, *II.6*

DEF_REF_LIST: **B.5.26**

Default behaviour proforma: *3.6.18*, *B.1*

Default behaviour: **3.6.18**, *3.6.19*, *3.6.22*, *15.1*, *15.2.1*, 15.4, *15.18.1*, *15.18.2*, *15.18.4*, *B.5.5*, B.5.5.4, *II.4*

Default duration: *11.12*

Default dynamic behaviour: *3.6.26*, 9.5

Default expansion: *15.18.3*

Default expression: *11.16.1*, *11.16.2*

Default group reference: **3.6.20**, *9.4*, *10.5*

Default group: **3.6.19**, *9.1*

Default identifier: **3.6.21**, *10.5*, *15.18.2*, *A.4.2.11*

Default index: 10.1, 10.5, *A.5.1*

Default library: *3.6.20*, **3.6.22**, *3.6.23*, *3.6.52*, *9.4*, *10.5*, *15.4.1*, *15.18.2*

Default objective: *10.5*

Default reference: **3.6.23**, *15.2.1*, 15.18.2, *B.5.5.4*

Default tree: *15.10.1*, *15.14*, *15.18.1*, *15.18.3*, *A.3.3.33*, *A.4.2.9*, *E.3.1*, *II.4*

Default value: *13.6*

DEFAULT: *11.3.3.3.1*, *15.18.1*, *A.4.2.5*

Default: *15.4.1*, 15.18, *15.18.2*, 15.18.6, *B.5.1*, *B.5.2.3*, *B.5.5.1*, *B.5.5.4*, *B.5.5.5*, II.4

Defect report: *B.2*

Definition by reference: 11.7

DEFINITIONS: *A.4.2.5*

DER: *11.15.2*, *11.15.4*, *11.15.5*, *13.4*, *14.4*

Derivation path: **3.6.24**, *13.4*, *13.6*, *14.3*, *d)*, *14.4*, *A.3.3.22*, *E.2.3*

Derivation: *A.1*

Detailed comments: *11.3.4.1*

Distinguished value: *A.4.2.6*

Distributed test method: *4.2*, II.3.3

DO: *11.3.4.8*, *A.4.2.4*

Done event: **15.9.10.2**

DONE: *15.9.10*, *15.9.10.2*, *A.4.2.4*, *B.5.7.2*, *B.5.12.2*, I.15

Dot notation: 15.10.2.2, 15.10.2.3

DS: **4.2**

Dynamic behaviour: *3.6.12*, *11.13.2*

Dynamic chaining: **3.6.25**, *12.4*

Dynamic part: **3.6.26**, 9.5, *11.1*, 15, *A.3.3.36.2*

E

EBDIF: *A.4.2.4*

Element: *15.10.3*

ELSE: *A.4.2.4*

EMBEDDED: *A.4.2.5*

Encoding definition: *11.3.3.2.1*, *11.15.2*, *11.15.4*, *11.15.5*, *11.16.1*, *11.16.2*, *11.16.4*

Encoding operation: *11.16.3*

Encoding rules precedence: 11.16.4

Encoding rules: *11.16.1*, *11.16.2*, *11.16.4*

Encoding variations: *11.2.3.2*, *11.2.3.3*, *11.2.3.4*, *11.2.3.5*, *11.15.2*, *11.15.4*, *11.15.5*, *11.16.2*, *13.2*, *13.4*, *14.2*, *14.4*

END: *11.3.4.4*, *A.4.2.4*, *A.4.2.5*

ENDCASE: *11.3.4.9*, *A.4.2.4*

ENDIF: *11.3.4.7*

ENDVAR: *11.3.4.3*, *A.4.2.4*

ENDWHILE: *11.3.4.8*, *A.4.2.4*

Enumerated type: *A.4.2.6*

ENUMERATED: *A.4.2.5*, *A.4.2.6*

Equivalence of TTCN forms: 5

ETS: **4.1**

EVAL_VERDICT_ENTRY: **B.5.23.1**

EVALUATE_BOOLEAN: **B.5.15.1**

EVALUATE_CONSTRUCT: **B.5.18.1**

EVALUATE_EVENT: **B.5.7.1**

EVALUATE_EVENT_LINE: **B.5.6.1**

EVALUATE_LEVELS: **B.5.4.1**

EVALUATE_PSEUDO_EVENT: **B.5.14.1**

EVALUATE_TEST_CASE: *B.1*, *B.5.3.1*, **B.5.4.1**

EVALUATE_TEST_COMPONENT: *B.1*, *B.5.3.1*, **B.5.20.1**

EVALUATE_TEST_SUITE: *B.1*, *B.5.2.3*, **B.5.3.1**

Evaluation tree: *B.1*, *B.5.2.1*, *B.5.2.3*

Event line: *15.9*, *15.10.1*, *15.10.4.1*, 15.10.6, *II.7*, *II.9*

Event matching: 15.10.6

EVENT_TYPE_OF: **B.5.26**

Examples of tabular constraints: I.1

Examples: Appendix I

EXCEPT: *A.4.2.5*

EXCLUDE_INCOMPATIBLE_ENTRY: **B.5.23.1**

Executable test case error: *6.5*

Executable test case: *6.5*

Executable test suite: *1*, *4.1*

EXECUTE_ASSIGNMENT: **B.5.16.1**

Execution of a test suite: B.5.3

EXPAND_ATTACHMENTS: **B.5.5.5**

EXPAND_CURRENT_LEVEL: **B.5.5.1**

EXPAND_REPEATS: **B.5.5.3**

EXPAND_SUBTREE: **B.5.5.5**

Expanded test suite: **3.6.27**

Expanding a set of alternatives: B.5.5.1

Expanding modularized test suite: B.4

Expansion of aliases: 11.21

Expansion of default trees: 15.13.7

Explicit external: **3.6.28**

EXPLICIT: *A.4.2.5*

Explicitly defined object: **3.6.29**, *3.6.32*, *3.6.33*, *3.6.36*

Explicitly exported object: **3.6.30**, *3.6.32*, *3.6.36*

Explicitly external object: *3.6.33*

Explicitly imported object: **3.6.31**, *3.6.32*, *3.6.36*, *3.6.37*, *3.6.39*, *10.7.1*

Explicitly imported: *B.1*

EXPORT: *A.4.2.5*

Export: *11.9*

Exported object: **3.6.32**

Exporting object type: 10.6

External object: *3.6.28*, **3.6.33**

External objects: *C.3.1*, **C.3.2**

EXTERNAL: *10.6*, *10.7.2*, *A.4.2.5*, *C.2.2*

Externally declared object: *3.6.35*

Externally defined object: *3.6.46*

F

F: *15.17.2*, *15.17.3*, *A.4.2.4*

FAIL: *15.17.1*, *15.17.2*, *15.17.3*, *15.17.4*, *15.18.1*, *A.4.2.4*, *B.5.23.2*

Fail: *3.6.54*

FALSE: *10.2*, *10.3*, *11.2.2*, *11.3.3.3.1*, *11.3.3.3.2*, *11.3.4.7*, *11.3.4.8*, *11.16.1*, *11.16.2*, *15.11*, *A.4.2.4*, *A.4.2.5*, *B.5.8.2*, *B.5.9.2*, *B.5.10.2*, *B.5.11.2*, *B.5.12.2*, *B.5.15.2*

FDT: **4.3**

Field encoding definition: *11.2.3.2*, *11.2.3.4*, *11.15.2*, *11.15.4*, *11.16.3*, *13.4*

Field: *15.10.3*

FIFO: **4.3**, *11.10*

Final verdict: *15.9.10.2*, *15.17.1*, *15.17.3*, *15.18.1*, *II.2*

FIRST_LEVEL: **B.5.25**

Formal Description Technique: *4.3*

Formal description technique: *1*

Formal parameter list: *12.3*, *13.4*, *13.7*, *d)*, 14.7, *14.7*, *15.9.1*, *15.16.2*, *A.4.2.11*, *A.4.2.12*

Formal parameter name precedence: *A.4.2.12*

Formal parameter: *A.4.2.14*

Formal parameters: *3.6.85*, *15.7.2*, *15.13.5*

Free text: 7.4

FROM: *A.4.2.5*

G

GeneralizedTime: *A.4.2.5*

GeneralString: *A.4.2.4*, *A.4.2.5*

Global result variable: **3.6.34**

Global test steps: 9.3.2

GOTO construct: **15.14**

GOTO: *15.2.1.3*, *15.6*, *15.8*, *15.9.5.1*, *15.14*, *15.17.1*, *A.4.2.4*, *B.1*, *B.5.5.1*, *B.5.18.2*, B.5.21, *B.5.21*, B.5.22

GOTO_NEXT_LEVEL_OR_STOP_WITH_ VERDICT: *B.5.21*, *B.5.22*, **B.5.25**

GraphicString: *A.4.2.4*, *A.4.2.5*

H

HEX_TO_INT: *11.3.3.2.1*, 11.3.3.2.2, *A.4.2.4*

HEXSTRING: **11.2.2**, *11.18.1*, *11.18.2*, *15.10.4.2*, *A.4.2.4*

Hyphen symbol: *11.15.4*

I

I: *15.17.2*, *15.17.3*, *A.4.2.4*

IA5String: *A.4.2.4*, *A.4.2.5*

IDENTIFIER: *A.4.2.5*

Idle testing state: *15.17.3*

IF THEN ELSE: *11.3.4.7*

IF THEN: *11.3.4.7*

IF: *A.4.2.4*

IF_PRESENT: *A.4.2.4*

IfPresent: *12.6.6.1*, **12.6.6.2**

Illegal variations of encoding: 11.16.3

Implementation Under Test: *4.1*

Implicit external: **3.6.35**

Implicit send event: **3.6.38**, **15.9.6**

IMPLICIT SEND: *15.6*, *15.8*, *15.9.5.3*, *15.9.6*, *15.16.1*, *15.17.1*, *B.5.7.2*, *B.5.13.2*, *II.3.5*

IMPLICIT: *A.4.2.5*

IMPLICIT_SEND: **B.5.13.1**

Implicitly exported object: *3.6.32*, **3.6.36**

Implicitly external object: *3.6.33*

Implicitly imported object: **3.6.37**, *3.6.37*, *3.6.39*, *10.7.1*, *B.1*

Import part: 9.5, 10.7.1, *C.1*

IMPORT: *A.4.2.5*

Import: 10.7.2, *C.3.1*, **C.3.3**

Imported object: *3.6.27*, *3.6.30*, *3.6.33*, **3.6.39**, *10.7.1*, *B.4*

INCLUDES: *A.4.2.5*

INCONC: *15.17.1*, *15.17.2*, *15.17.3*, *A.4.2.4*, *B.5.23.2*

Inconclusive verdict: *15.17.3*

Indentation: *3.6.61*, *15.2.5*, *15.6*, *15.9.5*, *15.15*, *A.5.1*, *A.5.2*, *B.5.5.3*

Index notation: 15.10.2.4

INFINITY: *11.2.3.2*, *11.14.2*, *11.15.2*, *11.17.2*, *11.18.2*, *12.6.4.6*, *12.6.6.1*, *A.4.2.4*

INPUT_Q: **B.5.26**

Inside values: 12.6.5

INSTANCE: *A.4.2.5*

INT_TO_BIT: *11.3.3.2.1*, 11.3.3.2.5, *A.4.2.4*

INT_TO_HEX: *11.3.3.2.1*, 11.3.3.2.4, *A.4.2.4*

INTEGER: **11.2.2**, *11.2.3.2*, *11.3.3.3.3*, *11.12*, *11.14.2*, *11.17.2*, *11.18.2*, *12.6.4.6*, *12.6.5.1*, *12.6.5.3*, *12.6.6.1*, *15.10.2.3*, *15.10.2.4*, *15.12.2*, *15.12.4*, *A.4.2.4*, *A.4.2.5*, *A.4.2.6*

INTERSECTION: *A.4.2.5*

Invalid field encoding definition: *14.2*, *14.4*

Invalid field encoding: *11.2.3.3*, *11.16.3*, *12.6.4.2*

Invalid test event: *15.17.4*

IS_CHOSEN: 11.3.3.3.2, *A.4.2.4*

IS_EXPANDED: **B.5.25**

IS_PRESENT: 11.3.3.3.1, *A.4.2.4*

IsDefault: *B.1*

IsExpanded: *B.1*

ISO646String: *A.4.2.5*

IUT: *3.6.13*, *3.6.38*, **4.1**, *10.2*, *11.10*, *11.11*, *11.15.1*, *15.9.6*, *A.4.2.4*, *II.3.4*, *II.4*

L

Label: *3.6.5*, *15.2.1.3*, **15.14**

Length: *11.18.2*, **12.6.6.1**

LENGTH_OF: 11.3.3.3.4, *A.4.2.4*

Level of indentation: **3.6.40**

LEVEL_OF: *B.5.2.5*, **B.5.25**

Levels of alternatives: B.5.2.5

Lifetime of events: 15.9.4

Line continuation: 15.2.5, *A.5.1*

Literal values: *11.16.1*, *11.16.2*

Local result variable: **3.6.41**, *B.5.20.2*

Local test method: *4.2*, II.3.2

Local test steps: 9.3.2

Local tree: **3.6.42**, *3.6.85*, *3.6.86*, *15.2.5*, *15.4.1*, *15.6*, *15.10.1*, *15.13.2*, *15.13.3*, *15.13.4.1*, *15.15*, *A.4.2.9*, *A.4.2.10*, *A.4.2.11*, *A.5.2*

Local variables: *11.3.4.3*, *11.3.4.4*, *11.3.4.6*

Location of object: 10.6

LOG procedure: **B.5.24.1**

Lower Tester Control Function: *4.1*

Lower tester: *4.1*, *11.13.1.2*

LS: **4.2**

LT: **4.1**, *11.9*, *11.10*, *15.2.1.3*, *15.8*, *15.9.1*, *15.9.5.1*, *15.9.6*, *15.9.7*, *A.4.2.4*, *B.5.8.2*, *B.5.9.2*, *B.5.10.2*, *B.5.11.2*, *B.5.12.2*, *II.4*

LTCF: **4.1**

M

Macro expansion: *12.2*, *13.2*, *13.4*, 15.10.3, *15.10.3*, *A.3.3.34*, *A.4.2.8*

Macro symbol: *11.14.3*, *11.15.3*

Main test component: *3.6.34*, **3.6.43**, *3.6.53*, *4.3*, *8.1*, 11.13.1.1, *11.13.1.3*, *15.9.10.1*, *B.5.2.3*, *B.5.3.1*, *B.5.23.2*

MAKE_TREE: **B.5.25**

Matching ASP: *12.6.1*

Matching attributes of values: 12.6.2

Matching inside values: 12.6.2

Matching instead of values: 12.6.2

Matching mechanism: *3.6.65*, *12.2*, *12.5*, 12.6.3, *14.1*, *15.9.9*

Matching mechanisms: 12.6.2

Matching PDU: *12.6.1*

Matching values in constraints: 12.6.1

MAX: *A.4.2.5*

Means of Testing: *4.1*, *II.5*

MIN: *A.4.2.5*

min: *A.4.2.4*

MOD: *A.4.2.4*

Modified ASN.1 constraints: 14.6, *14.7*, 14.7

Modified constraints: *3.6.7*, *3.6.24*, **3.6.44**, 13.6, *13.7*, *A.3.3.19*, *A.3.3.22*, *E.2.3*, *E.2.4*, I.1.2.5, *I.2.2.5*, I.3, *II.6*

Modular TTCN: I.14

Modularized test suite: **3.6.45**

Module constraints part: *C.1*

Module declarations part: *C.1*

Module default index: *C.2.1*, **C.2.6**

Module dynamic part: *C.1*

Module exports: C.2.2

Module import part: C.3

Module structure: C.2.3

Module test case index: *C.2.1*, **C.2.4**

Module test step index: *C.2.1*, **C.2.5**

Module: *3.6.28*, *3.6.29*, *3.6.32*, *3.6.33*, **3.6.46**, *3.6.50*, *3.6.69*, *10.7.1*, *B.1*, *B.4*

MOT: **4.1**, *15.9.5.3*

MPyT: *3.6.34*

ms: *A.4.2.4*

MTC: *3.6.58*, **4.3**, *8.1*, *8.2*, *11.8.1*, *11.8.3*, *11.13.1.2*, *11.13.2*, *15.2.4*, *15.9.10.2*, *15.17.5*, *15.18.7*

MTC_R: *3.6.58*, *15.17.5*

Multi-party testing: I.10

Multiplexing/demultiplexing: I.11

Multi-protocol test cases: I.13

MuxValue: 11.10, *I.11*

N

NEW_LABEL: **B.5.25**

Non-concurrent test case: **3.6.47**

none: *A.4.2.4*

NOT: *A.4.2.4*

ns: *A.4.2.4*

NULL: *A.4.2.5*

NUMBER_OF_ELEMENTS: 11.3.3.3.3, *A.4.2.4*

NumericString: *A.4.2.4*, *A.4.2.5*

O

Object group: 7.3.2

Object name: 7.3.2, 7.3.3

OBJECT: *A.4.2.5*

Object: **3.6.48**, *3.6.50*, *10.7.2*

OBJECT_MATCHES: **B.5.9.1**

ObjectDescriptor: *A.4.2.5*

OBJECTIDENTIFIER: **11.2.2**, *A.4.2.4*

OCTET: *A.4.2.5*

OCTETSTRING: **11.2.2**, *11.3.3.3.4*, *11.18.1*, *11.18.2*, *15.10.4.2*

OF: *A.4.2.4*

Omit symbol: *12.5*

OMIT: *10.7.2*, *14.6*, *A.4.2.4*

Omit: **12.6.4.2**

Open Systems Interconnection: *4.3*

Operational semantics: *1*, **3.6.49**, *5*, *6*, *15.9.5.2*, Annex B, B.5

OPTIONAL: *11.3.3.3.1*, *11.3.3.3.3*, *12.5*, *12.6.4.2*, *12.6.6.2*, *14.5*, *14.8*, *A.4.2.5*

OR: *A.4.2.4*

Order of receipt of events: *15.9.5.4*

Original source object: **3.6.50**

OSI: *1*, *2*, **4.3**, *A.4.2.1*

Otherwise event: **3.6.51**, *15.9.7*

OTHERWISE function: **B.5.10.1**

OTHERWISE: *3.6.91*, *15.8*, *15.9.5.3*, *15.9.7*, *15.9.8*, *15.10.6*, *15.17.4*, *15.18.5*, *A.3.3.33*, *A.4.2.4*, *B.5.7.2*, *B.5.10.2*, *B.5.15.2*

OUTPUT_Q: **B.5.26**

Overview part: **3.6.52**

P

P: *15.17.2*, *15.17.3*, *A.4.2.4*

Page continuation: 16, 16.1, 16.2, *A.5.1*

Parallel test component: *3.6.43*, **3.6.53**, *4.3*, *8.1*, 11.13.1.2, *11.13.1.3*, *15.9.10.1*, *B.5.2.3*, *B.5.3.1*, *B.5.23.2*

Parameter list: *12.3*, 13.5, 13.7, 14.7, *15.2.1*, 15.7, *15.9.1*, *15.13.4.1*, *15.16.2*, *15.18.2*, *A.3.3.19*, *A.3.3.22*, *E.2.3.2*

Parameter: *3.6.13*, *3.6.66*, *3.6.68*, d), *11.15.2*, *11.19*, 13.5, *14.5*, *15.9.4*, *15.10.3*, *A.3.3.19*, *A.3.3.22*, *A.3.3.34*, *A.4.2.7*, *II.6*

Parameterization: *3.6.25*, *11.1*, *11.4*, *15.18.2*, */* STATIC SEMANTICS* -, *A.3.3.19*, *A.3.3.22*, *A.3.3.23*

Parameterized compact constraints: *E.2.3.2*

Parameterized constraint: *3.6.7*, 12.3, 13.5, *A.4.2.11*, I.1.2.4, I.1.2.5, I.2.2.4

PASS: *15.17.1*, *15.17.2*, *15.17.3*, *15.17.4*, *A.4.2.4*, *B.5.23.2*

Pass: *3.6.54*

Passing of constraints: 15.13.5

Passing parameters: 15.16.2

PCO declaration: 11.10, *SYNTAX DEFINITION:*, *11.15.2*

PCO model: *15.9.1*

PCO queue: *15.9.2*

PCO type: 11.9, *SYNTAX DEFINITION:*, *11.15.2*, *12.3*, *15.7.2*

PCO: *3.6.57*, *3.6.60*, *3.6.72*, *3.6.73*, **4.1**, *8.1*, *8.2*, *9.5*, *10.6*, *11.3.4.1*, *11.9*, *11.10*, *11.11*, *SYNTAX DEFINITION:*, *11.13.1.1*, *11.13.1.3*, *11.13.2*, *11.14.2*, *11.14.4*, *11.14.5*, *11.15.1*, *11.15.2*, *11.15.4*, *11.15.5*, *15.2.4*, *15.3.1*, *15.4.1*, *15.9*, *15.9.1*, *15.9.5.3*, *15.9.5.4*, *15.9.6*, *15.9.7*, *15.9.8*, *15.9.10.1*, *15.18.1*, *15.18.8*, *A.4.2.13*, *B.1*, *B.5.4.2*, *B.5.8.2*, *B.5.9.2*, *B.5.10.2*, *B.5.11.2*, *B.5.12.2*, *I.11*, II.10

PDU constraint compact proforma: *E.2.3*

PDU constraint declaration: *3.6.62*, *13.2*, 13.4, *A.5.1*

PDU constraints: 7.3.4, *11.16.3*, *12.6.6.1*, *13.4*, *14.1*

PDU field value: *11.20*, *12.2*, *12.4*, *12.6.4.5*, *12.6.4.6*, *15.9.3*, *15.9.4*

PDU field: *3.6.66*, *11.2.1*, *11.16.3*, *11.17.1*, *12.1*, *12.5*, *12.6.2*, 12.6.3, *12.6.4.1*, *12.6.4.2*, *12.6.4.3*, *12.6.4.4*, *12.6.4.5*, *12.6.4.7*, *12.6.4.8*, *12.6.5.1*, *12.6.5.3*, *12.6.6.2*

PDU identifier: *11.15.2*, *11.21*, *15.9.1*

PDU specification in ASN.1: 11.15.5

PDU type definition: *3.6.3*, *3.6.68*, 11.15, 11.19, 11.20, *13.4*, *E.2.3*, I.4, *II.6*

PDU type: *11.3.4.2*, *11.8.1*, *11.8.3*, *13.4*, *14.4*, *15.7.2*

PDU: *3.6.1*, *3.6.9*, *3.6.13*, *3.6.25*, *3.6.38*, *3.6.44*, *3.6.57*, *3.6.60*, *3.6.66*, *3.6.68*, **4.3**, *7.3.1*, *9.5*, *11.2.1*, *11.2.2*, *11.2.3.2*, *11.2.3.3*, *11.2.3.4*, *11.2.3.5*, *11.3.4.1*, *11.3.4.2*, *11.6*, *11.7*, *11.10*, *11.14.2*, *11.15.1*, *11.15.2*, *11.15.3*, *11.15.4*, *11.15.5*, *11.16.2*, *11.16.4*, *11.17.1*, *11.17.2*, *11.17.3*, 12.6.3, *13.2*, *13.6*, *14.5*, *14.6*, *14.8*, *15.9*, *15.9.5.3*, *15.9.5.4*, *15.9.6*, *15.10.1*, *15.10.2.2*, *15.10.2.3*, *15.10.3*, *15.10.4.1*, *15.10.6*, *15.16.1*, *15.18.8*, *A.3.3.19*, *A.3.3.22*, *A.3.3.34*, *A.4.2.4*, *A.4.2.5*, *A.4.2.7*, *A.4.2.8*, *B.5.2.3*, *B.5.8.2*, *B.5.9.2*, *B.5.10.2*, *B.5.11.2*, *B.5.12.2*, *B.5.13.2*, *B.5.16.2*, *E.2.1*, *II.3.1*

PERMUTATION: *A.4.2.4*

Permutation: **12.6.5.3**, *12.6.6.1*

PICS proforma: *11.4*

PICS: *3.6.80*, *3.6.81*, **4.1**, *10.2*, *11.4*, *11.6*, *11.7*, *11.12*, *C.2.2*

PIXIT proforma: *11.4*

PIXIT: *3.6.80*, *3.6.81*, **4.1**, *10.2*, *11.4*, *11.6*, *11.7*, *11.10*, *11.12*, *15.9.6*, *C.2.2*

Point of attachment: *15.13.5*

Point of control and observation: *4.1*, *8.1*

Postamble: *II.2*

Preamble: *II.2*

Precautions for concurrent TTCN: 15.9.5.4

Precedence of assignments and qualifiers: 15.10.6

Precedence of operators: Table 3

Precedence of pseudo-events: 15.11

Precedence: *15.17.2*, *A.4.2.11*, B.2, *II.8*

Predefined type: *11.3.4.2*, *d)*, *11.6*, *11.7*, *11.8.1*, *11.15.2*, *11.16.3*

Predefined variable: *3.6.41*

Preliminary result variable: *B.5.4.2*

Preliminary result: *3.6.34*, *3.6.41*, **3.6.54**, *3.6.58*, *11.13.1.1*, *11.13.1.2*, *15.9.10.2*, *15.17.1*, 15.17.2

PRESENT: *A.4.2.5*

PrintableString: *A.4.2.5*

PRIVATE: *A.4.2.5*

Procedural definition of test suite operation: *A.4.2.14*

Procedural definition: *11.3.4.3*

Procedure statement: *11.3.4.4*

Protocol Data Unit: *1*, *4.3*

Protocol error: *15.17.2*

Protocol Implementation Conformance Statement: *4.1*

Protocol Implementation eXtra Information for Testing: *4.1*

ps: *A.4.2.4*

Pseudo-code keywords: B.5.2.1

Pseudo-code notation: B.5.2

Pseudo-code precedence: *B.2*

Pseudo-code procedures and functions: B.5.2.2

Pseudo-code process: B.5.2.3

Pseudo-code with natural language: B.5.2.4

Pseudo-code: *B.5.2.3*, *B.5.2.4*, *B.5.5.4*

Pseudo-event: **3.6.55**, *3.6.61*, *3.6.90*, *15.8*, 15.11, *B.5.1*, *B.5.5.4*, B.5.14, *B.5.14.2*

PTC: *3.6.58*, **4.3**, *8.1*, *8.2*, *11.13.1.1*, *11.13.1.2*, *11.13.2*, *15.2.4*, *15.9.10.1*, *15.9.10.2*, *15.17.5*, *15.18.7*

Q

Qualified event: **3.6.56**

Qualifier evaluation: 15.10.6

Qualifier: *15.6*, *15.8*, *15.9.2*, *15.10.4.1*, 15.10.5, *15.11*, *15.15*, 15.16.3

Queue: *15.9.2*

R

R: *3.6.58*, *15.17.2*, *15.17.3*, *15.17.5*, *15.18.1*, *B.5.23.2*, *II.2*

R_TYPE: **11.2.2**, *15.17.2*, *15.17.5*

R_Type: *A.4.2.4*

Range: *11.18.2*, **12.6.4.6**, *12.6.6.1*

READ_TIMER: **B.5.17.1**

READTIMER operation: **15.12.4**

READTIMER: *15.12.1*, *15.12.4*, *A.4.2.4*, *B.5.14.2*, *B.5.17*

REAL: *A.4.2.5*

Receive event: **3.6.57**, 11.20, *12.1*, *12.2*, 15.9.2, *15.10.4.1*, *A.3.3.33*

RECEIVE function: **B.5.9.1**

RECEIVE: *8.1*, *8.2*, *11.16.4*, *15.9.4*, *15.9.5.3*, *15.9.6*, *15.10.6*, *15.16.1*, *B.5.7.2*, *B.5.9.2*, *B.5.15.2*

ReceiveObject: *B.5.2.3*

Record references: 15.10.2.2

Recursive tree attachment: 15.13.6

References in chaining of constraints: 15.10.2.2

References using tables: 15.10.3

RELABEL: **B.5.25**

Relational operators: 11.3.2.3

Remote test method: *3.6.38*, *4.2*, *15.9.6*, II.3.5

REMOVE_OBJECT: **B.5.9.1**

REPEAT construct: **15.15**

REPEAT: *15.6*, *15.8*, *15.15*, *15.17.1*, *A.4.2.4*, *B.5.1*, *B.5.5*, *B.5.5.1*, B.5.5.3, *B.5.5.5*, *B.5.18.2*

RepeatTree: *B.5.5.3*

REPLACE: *14.6*, *A.4.2.4*

REPLACE_ALT_TREE: **B.5.25**

REPLACE_PARAMETERS: **B.5.25**

Restrictions on using events: 15.9.5.3

Result type: *11.3.4.5*

Result variable: **3.6.58**, *3.6.58*

RETURN statement: **15.18.3**

RETURN: *15.18.1*, *15.18.3*, 15.18.6, *15.18.6*, *B.1*, *B.5.2.3*, *B.5.18.2*, *B.5.22*

ReturnDefaults: *B.5.2.3*

ReturnLevel: *B.5.2.3*

RETURNVALUE: *11.3.4.1*, *11.3.4.5*, *A.4.2.4*

Root tree: **3.6.59**, *15.6*, *15.13.3*, *15.13.4.1*, *15.14*, *15.18.5*, *A.4.2.9*, *A.5.2*

ROOT_TREE: **B.5.25**

RS: **4.2**

S

SAP: **4.3**, *11.10*, *II.10*

SAVE_DEFAULTS: **B.5.5.2**

Scope of tree attachment: 15.13.2

Scoping rules: *15.13.4.1*

sec: *A.4.2.4*

Selection expression: *3.6.52*, 11.5, *I.7*

Selection: *11.1*, *b)*, *I.7*

Semantics of TTCN: B.1

Send event: **3.6.60**, 11.19, *12.1*, *12.2*, 15.9.3, *15.10.4.1*, B.5.8, II.9

SEND function: **B.5.8.1**

SEND: *8.1*, *8.2*, *11.10*, *12.5*, *15.9.4*, *15.10.6*, *B.5.7.2*, *B.5.15.2*

SEND_EVENT: **B.5.8.1**

SendObject: *B.5.2.3*

SEQUENCE OF INTEGER: *12.6.5.1*, *12.6.5.3*

SEQUENCE OF: *11.3.3.3.3*, *11.18.2*, 12.6.3, *12.6.5.1*, *12.6.5.2*, *12.6.5.3*

SEQUENCE: 12.6.3, *14.5*, *14.8*, *15.10.2.2*, *15.10.2.3*, *15.10.2.4*, *A.4.2.5*

Service Access Point: *4.3*

Set of alternatives: **3.6.61**, *15.6*, *15.9.5.2*, *15.9.9*, *15.13.4.1*, *15.18.5*, *A.3.3.33*, *B.5.5.4*, *B.5.5.5*

SET OF: *11.3.3.3.3*, *11.18.2*, *12.5*, 12.6.3, *12.6.4.7*, *12.6.4.8*, *12.6.5.1*, *12.6.5.2*, *12.6.6.1*

SET: *12.5*, 12.6.3, *14.5*, *14.8*, *15.10.2.2*, *15.10.2.3*, *A.4.2.5*

Simple CMs: 11.17.1

Simple type: 11.2.3.2, *11.6*, *11.7*, *11.14.2*, *11.14.3*, *11.15.2*, *11.15.3*

Single constraint table: **3.6.62**, *13.1*, *E.1*, *E.2.1*, *E.2.4*

SIZE: *A.4.2.5*

Snapshot semantics: **3.6.63**, 15.9.5.2

SNAPSHOT: *B.5.12.2*

SNAPSHOT_FIXED: **B.5.26**

Specific value: **3.6.65**, *12.2*, 12.6.3, *12.6.4.5*, *12.6.6.1*, *15.9.3*

Splitting and Recombining: I.12

SPyT: *3.6.34*

Stable testing state: *15.17.3*

Standardized ATS: *6.5*, *II.8*

START operation: **15.12.2**

START: *15.12.1*, *15.12.2*, *A.4.2.4*, *B.5.14.2*

START_TIMER: **B.5.17.1**

STATEMEMT_LINE_TYPE_OF: **B.5.26**

Statement line: *B.1*

StatementLine: *B.5.2.5*

Static chaining: **3.6.66**, *12.4*

Static conformance requirements: *1*

STATIC SEMANTICS: *A.4.1*

Static semantics: **3.6.67**, *5*, Annex A, 390, *B.1*

STATIC: *11.3.4.3*, *A.4.2.4*

Step-wise expansion: *B.5.2.1*

STOP_TEST_CASE: **B.5.26**

STRING: *A.4.2.5*

Structure: *15.10.3*

Structured type constraint declaration: 13.2

Structured type constraints: 7.3.4, *A.4.2.15*, E.2.4

Structured type: *3.6.9*, **3.6.68**, 11.2.3.3, *11.2.3.3*, *11.14.2*, *11.14.3*, *11.15.2*, *11.15.3*, *11.18.1*, *11.20*, *12.6.1*, 12.6.3, *13.1*, *13.2*, *13.4*, *15.10.3*, *A.3.3.19*, *A.3.3.22*, *A.4.2.8*, *E.2.1*, *E.2.4*, II.6

Structured types within ASP type: 11.14.3

Style guide: Appendix II

Submodule: **3.6.69**

Subsequent behaviour: *15.13.3*

SUBSEQUENT_BEHAVIOUR_TO: **B.5.25**

SUBSET: *A.4.2.4*

SubSet: **12.6.4.8**, *12.6.6.1*

Substructure: *3.6.68*, *11.20*, *13.3*, *15.10.3*, *A.3.3.19*, *A.3.3.22*, *A.3.3.34*

Subtree: *B.5.5.4*, *II.4*

Suite overview part: 9.5

Suite overview: *10.1*

SUPERSET: *A.4.2.4*

SuperSet: **12.6.4.7**, *12.6.6.1*

SUT: **4.1**

Syntactic metanotation: A.2.1

Syntax definition: 5

Syntax forms of TTCN: 5

Syntax production: *5*, A.3

SYNTAX: *A.4.2.5*

System Under Test: *4.1*

T

T61String: *A.4.2.5*

Tabular ASP type definition: *13.1*

Tabular PDU type definition: *13.1*

TAKE_SNAPSHOT: **B.5.26**

TCP: **4.3**

TeletexString: *A.4.2.5*

TERMINATE_TEST_CASE: **B.5.26**

Test body: *II.2*

Test case dynamic behaviour: *7.3.1*, 7.3.4, 9.5, 15.2, *15.18.2*, *A.5.1*, *A.5.2*, E.3, E.3.2

Test case error processing: B.3

Test case error: *11.3.3.2.4*, *11.3.3.2.5*, *11.16.1*, *11.16.2*, *15.9.3*, *15.9.10.1*, *15.12.2*, *15.17.3*, *B.5.4.2*

Test case execution pseudo-code: B.5.4.1

Test case execution, natural language: B.5.4.2

Test case identifier: **3.6.70**

Test case index: 10.1, 10.3, *b)*, *A.5.2*

Test case root tree: *15.7.2*

Test case selection expression: 10.2

Test case selection: *11.1*, *d)*, b)

Test case termination: 11.8.4

Test case variable: *3.6.34*, *3.6.58*, **3.6.71**, *7.3.1*, *11.6*, *11.7*, *11.8.1*, 11.8.3, *11.8.4*, *11.12*, *12.3*, *15.10.1*, *15.10.4.1*, *15.13.1*, *15.17.2*, *B.5.20.2*

Test case writer: II.5

Test case: *1*, *3.6.10*, *3.6.11*, *3.6.12*, *3.6.23*, *3.6.26*, *3.6.34*, *3.6.47*, *3.6.52*, *3.6.54*, *3.6.59*, *3.6.61*, *3.6.63*, *3.6.70*, *3.6.71*, *3.6.73*, *3.6.74*, *3.6.82*, *3.6.89*, *9.1*, *9.2*, *9.3.1*, *9.5*, *10.2*, *10.3*, *d)*, *b)*, *11.8.3*, *11.8.4*, *15.1*, *15.2.1*, *15.3.1*, *15.4.1*, *15.9.5.1*, *15.9.10.1*, *15.12.1*, *15.12.4*, *15.13.2*, *15.14*, *15.17.2*, *15.18.1*, *15.18.4*, *A.4.2.13*, *B.5.2.1*, *B.5.2.3*, *B.5.3.1*, B.5.4, *E.3.2*, II.2, *II.5*, *II.8*

Test component configuration declaration: 8.1

Test component configuration: *3.6.12*, *3.6.16*, *3.6.43*, **3.6.73**, 8.2, *11.13.1.3*, 11.13.2, *15.2.4*, *A.4.2.13*

Test component declaration: 8.1, 11.13.1.3

Test component: *3.6.12*, *3.6.15*, *3.6.16*, *3.6.41*, *3.6.43*, *3.6.53*, **3.6.72**, *3.6.73*, *11.12*, *15.9.10.2*

Test coordination procedures: *4.3*

Test event: *3.6.5*, *3.6.6*, *3.6.91*, *15.8*, 15.9, *15.10.4.1*, *A.5.1*

Test group identifier: *A.5.1*, *A.5.2*

Test group objective: 10.2, *C.2.3*

Test group reference: **3.6.74**, 9.2, *10.2*, *10.3*, *15.2.1*, *A.5.1*, *C.2.3*

Test group: *3.6.10*, *3.6.52*, *9.1*, *9.2*, *10.2*, *10.3*, *A.5.2*, *C.2.3*, *E.3.1*

Test laboratory: *6.5*

Test management protocol: *4.1*

Test method: II.3

Test outcome: *3.6.91*

Test purpose: *15.2.1*, *II.2*, *II.8*

Test realizer: *II.5*

Test result: *3.6.54*

Test step dynamic behaviour: *3.6.78*, 9.5, 15.3, *15.18.2*

Test step group reference: **3.6.76**, *9.3.2*, *10.4*

Test step group: **3.6.75**, *9.1*, *9.3.1*, *10.4*

Test step identifier: **3.6.77**, *10.4*, *A.4.2.11*

Test step index: 10.1, 10.4, *A.5.1*

Test step library: *3.6.52*, *3.6.76*, **3.6.78**, *3.6.84*, *9.3.1*, *9.3.2*, *10.4*, *15.3.1*, *15.13.2*, *15.13.3*, *15.15*, *15.18.5*, *A.4.2.10*, *II.2*

Test step objective: **3.6.79**, *10.4*, *15.3.1*

Test step root tree: *15.7.2*

Test step: *3.6.2*, *3.6.8*, *3.6.23*, *3.6.26*, *3.6.75*, *3.6.76*, *3.6.77*, *3.6.79*, *3.6.84*, *3.6.87*, *9.1*, *9.3.1*, *9.3.2*, *10.4*, *15.1*, *15.2.3*, *15.3.1*, *15.4.1*, *15.9.5.1*, *15.9.10.1*, *15.13.2*, *15.13.3*, *15.13.4.1*, *15.13.5*, *15.15*, *15.18.1*, *15.18.5*, *A.4.2.12*, *B.5.5.5*

Test suite constant: **3.6.80**, *11.2.1*, *b)*, 11.6, *11.6*, *11.7*, *11.14.2*, *11.15.2*, *12.3*, *B.5.2.3*

Test suite constants: *11.16.1*, *11.16.2*, *11.17.2*, *15.10.1*

Test suite exports: 10.1

Test suite index: *D.1*, **D.2.2**

Test suite operation description: 11.3.4

Test suite operation procedural definition: 11.3.4

Test suite operation, assignment: 11.3.4.6

Test suite operation, CASE: 11.3.4.9

Test suite operation, IF: 11.3.4.7

Test suite operation, parameter passing: 11.3.4.2

Test suite operation, RETURNVALUE: 11.3.4.5

Test suite operation, variables: 11.3.4.3

Test suite operation, WHILE: 11.3.4.8

Test suite operation: *11.3.4.2*, *11.3.4.3*, *11.16.3*, *A.4.2.14*

Test suite operations: I.6

Test suite parameter: **3.6.81**, *11.2.1*, 11.4, *b)*, *11.15.2*, *11.16.1*, *11.16.2*, *11.17.2*, *12.3*, *15.10.1*, *B.5.2.3*, *I.7*

Test suite parameters: *11.14.2*

Test suite specifier: *15.9.5.1*, *II.1*, *II.2*, *II.4*

Test suite structure: 9, 10.1, 10.2, *15.2.1*, *A.5.1*, *A.5.2*, *I.7*

Test suite type definition: 11.2, *11.15.2*, *12.6.6.1*

Test suite type: *11.2.3.4*, *11.3.4.1*, *11.3.4.2*, *11.8.1*, *11.8.3*, *11.14.2*, *11.16.3*, *11.17.2*, *14.2*

Test suite variable: **3.6.82**, *11.2.1*, *11.6*, *11.7*, 11.8.1, *11.8.1*, *11.8.2*, *11.8.3*, *11.12*, *11.13.1.1*, *11.13.1.2*, *12.3*, *15.10.4.1*, *15.13.1*, *B.5.2.3*

Test suite: *3.6.4*, *3.6.13*, *3.6.17*, *3.6.22*, *3.6.26*, *3.6.27*, *3.6.29*, *3.6.32*, *3.6.45*, *3.6.48*, *3.6.50*, *3.6.52*, *3.6.71*, *3.6.78*, *3.6.80*, *3.6.81*, *3.6.82*, *3.6.91*, *9.1*, *9.2*, *10.1*, *10.7.1*, *11.2.1*, *11.2.3.2*, *11.4*, *11.12*, *11.15.2*, *15.12.4*, *A.4.2.6*, *A.4.2.10*

Test system: *12.1*

Test verdict: *3.6.34*, *3.6.43*

Textual substitution: *15.13.4.1*, *B.5.20.2*

THEN: *A.4.2.4*

Timeout event: **3.6.83**, **15.9.9**

TIMEOUT function: **B.5.11.1**

TIMEOUT: *15.8*, *15.9.5.2*, *15.9.5.3*, *15.9.9*, *15.12.3*, *A.3.3.33*, *A.4.2.4*, *B.5.7.2*, *B.5.11.2*, *B.5.15.2*, *II.5*

Timer declaration: 11.12

Timer management: 15.12

Timer name: *15.9.9*

Timer operation: *3.6.55*, *15.8*, *15.11*, 15.12.1, B.5.17

Timer value: *15.12.2*

Timer: *3.6.83*, *15.9.9*, *II.5*

TIMER_EXPIRED: **B.5.11.1**

TIMER_OP_TYPE_OF: **B.5.26**

TIMER_OPS: **B.5.17.1**

TMP: **4.1**, *10.2*

TO: *11.18.2*, *12.6.4.6*, *A.4.2.4*

Transfer syntax: *A.1*

Transformation algorithm: *B.1*

Tree and Tabular Combined Notation: *4.2*

Tree attach symbol: *15.13.3*

Tree attachment: **3.6.84**, *15.4.1*, *c)*, 15.13, *15.13.1*, 15.13.2, 15.13.3, *NOTE* - 15.18.5, 15.18.6, B.5.5.5, *II.2*, *II.5*

Tree header: **3.6.85**, *A.4.2.10*, *A.4.2.11*

Tree identifier: *3.6.85*, **3.6.86**, *A.4.2.10*

Tree leaf: **3.6.87**

Tree name: 15.7

Tree node: **3.6.88**

Tree notation: **3.6.89**, *15.2.1.3*, 15.6

TreeReference: *B.5.5.3*

Trees with parameters: 15.7.2

TRUE: *10.2*, *10.3*, *11.2.2*, *11.3.3.3.1*, *11.3.3.3.2*, *11.3.4.7*, *11.3.4.8*, *b)*, *11.16.1*, *11.16.2*, *15.6*, *15.10.5*, *15.10.6*, *15.11*, *15.12.1*, *15.15*, *A.4.2.5*, *B.5.8.2*, *B.5.9.2*, *B.5.10.2*, *B.5.11.2*, *B.5.12.2*, *B.5.15.2*

TTCN ASP constraints: *A.4.2.15*

TTCN CM constraints: *A.4.2.15*

TTCN expression: *3.6.55*, 15.10

TTCN graphical form: *4.3*

TTCN machine: *B.1*, *B.5.2.3*, *B.5.3.1*

TTCN machine-processable form: *4.3*

TTCN module exports: *C.2.1*

TTCN module overview part: *C.1*, C.2

TTCN module structure: *C.2.1*

TTCN object: *7.3.1*, *7.3.2*, *7.3.3*, *7.3.4*

TTCN operations: 11.3

TTCN operators: 11.3

TTCN PDU constraints: *A.4.2.15*

TTCN semantics: *B.5.2.1*

TTCN statement: *3.6.2*, *3.6.6*, *3.6.18*, *3.6.61*, *3.6.87*, *3.6.88*, **3.6.90**, *15.2.1.3*, *15.2.3*, *15.5*, *15.6*, 15.8, *15.16.1*, *B.5.1*

TTCN type: 11.2

TTCN.GR: **4.3**, *5*, *6*, *7.1*, *7.3.5*, *7.4*, *15.6*, *A.1*, *A.4.1*, A.5

TTCN.MP: **4.3**, *5*, *6*, *7.1*, *7.4*, *11.2.3.4*, *11.14.4*, *11.15.4*, *14.1*, *15.6*, *A.1*, *A.4.1*, A.5, *E.1*, I.8

TTCN: **4.2**

Type definition using macros: I.4

Type definitions using ASN.1: 11.2.3.4

Type list: *11.16.3*

Type: *11.16.3*

TYPEIDENTIFIER: *A.4.2.5*

U

Unbound variable: *3.6.65*, *15.10.4.1*

Unbound variables: *11.3.4.3*

Underscore symbol: *11.14.4*, *11.15.4*

Unforeseen test event: *3.6.51*, **3.6.91**

Unforeseen test events: 15.9.7

UNION: *A.4.2.5*

UNIQUE: *A.4.2.5*

Units of length: 11.18.2

UNIVERSAL: *A.4.2.5*

UniversalString: *A.4.2.5*

Unqualified event: **3.6.92**

UNTIL: *A.4.2.4*

UPDATE_PRELIM: **B.5.23.1**

Upper tester: *4.1*, *11.13.1.2*

us: *A.4.2.4*

Use of REPEAT: I.5

UT: **4.1**, *11.9*, *11.10*, *15.2.1.3*, *15.9.1*, *15.9.5.1*, *15.9.7*, *A.4.2.4*, *B.5.8.2*, *B.5.9.2*, *B.5.10.2*, *B.5.11.2*, *B.5.12.2*, *II.4*

UTCTime: *A.4.2.5*

V

Value: *11.3.4.2*

ValueList: **12.6.4.5**

VAR: *11.3.4.3*

Variable declaration: *A.4.2.14*

Variable name: *A.4.2.14*

Variables: *11.3.4.3*

Verdict assignment: 15.17.5

Verdict: *3.6.5*, *11.13.1.1*, *15.2.1.3*, *15.2.3*, 15.17, B.5.22, *B.5.23.2*, *II.2*

VideotexString: VisibleString: *A.4.2.5*

VisibleString: *A.4.2.5*

W

WHILE: *A.4.2.4*

Wildcards: *12.5*

WITH: A.4.2.5

ITU-T RECOMMENDATIONS SERIES

Series A	Organization of the work of the ITU-T
Series B	Means of expression: definitions, symbols, classification
Series C	General telecommunication statistics
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	**Data networks and open system communications**
Series Y	Global information infrastructure
Series Z	Programming languages