INTERNATIONAL STANDARD

ISO/IEC 9646-3

First edition
1992-10-01

Information technology — Open Systems Interconnection — Conformance testing methodology and framework —

Part 3:
The Tree and Tabular Combined Notation (TTCN)

Technologies de l'information — Interconnexion de systèmes ouverts — Essais de conformité — Méthodologie générale et procédures —

Partie 3: Notation combinée arborescente et tabulaire (TTCN)

Reference number
ISO/IEC 9646-3:1992(E)

Contents

Page

Annexes

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 9646-3 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*.

ISO/IEC 9646 consists of the following parts, under the general title *Information technology — Open Systems Interconnection — Conformance testing methodology and framework*:

- *Part 1: General concepts*
- *Part 2: Abstract test suite specification*
- *Part 3: The Tree and Tabular Combined Notation (TTCN)*
- *Part 4: Test realization*
- *Part 5: Requirements on test laboratories and clients for the conformance assessment process*
- *Part 6: Protocol profile test specification*
- *Part 7: Implementaion conformance statement — Requirements and guidance on ICS and ICS proformas*

Annexes A, B and C form an integral part of this part of ISO/IEC 9646. Annexes D, E, F, G and H are for information only.

Introduction

This part of ISO/IEC 9646, one of a multi-part International Standard 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 ISO/IEC 9646.

In the abstract testing methodology a test suite is looked upon as a hierarchy ranging from the complete test suite, through test groups, test cases and test steps, down to test events. TTCN provides a naming structure to reflect the position 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.

This part of ISO/IEC 9646 is also to be published by CCITT as Recommendation X.292 (1992).

Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation (TTCN)

1 Scope

This part of ISO/IEC 9646 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 ISO/IEC 9646-1 and ISO/IEC 9646-2.

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

This part of ISO/IEC 9646 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 part of ISO/IEC 9646, although those test cases may need to be included in a conformance test suite standard.

NOTE 1 - For example, some static conformance requirements related to an application service may require testing techniques which are specific to that particular application.

This part of ISO/IEC 9646 specifies requirements on what a test suite standard may specify about a conforming realization of the test suite, including the operational semantics of TTCN test suites.

NOTE 2 - ISO/IEC 9646-4 specifies requirements concerning test realization including ETS derivation.

This part of ISO/IEC 9646 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 part of ISO/IEC 9646:

a) the specification of conformance test suites for multi-peer or Physical layer protocols;

b) the relationship between TTCN and formal description techniques;

c) the specification of test cases in which more than one behaviour description is to be run concurrently;

NOTE 3 - Use of parallel trees and synchronization between them is to be covered by a future amendment to this part of ISO/IEC 9646.

d) the means of realization of executable test suites (ETS) from abstract test suites.

2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 9646. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO/IEC 9646 are encouraged to investigate the possibility of applying the most recent editions of the standards listed below. Members of ISO and IEC maintain registers of currently valid International Standards.

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

ISO 7498 : 1984, *Information processing systems - Open Systems Interconnection - Basic Reference Model.* (See also CCITT Recommendation X.200 : 1988.)

ISO/TR 8509 : 1987, *Information processing systems - Open Systems Interconnection -Service conventions.*
(See also CCITT Recommendation X.210 : 1988.)

ISO/IEC 8824 : 1990, *Information technology - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1).*
(See also CCITT Recommendation X.208 : 1988.)

ISO/IEC 8825 : 1990, *Information technology - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).*
(See also CCITT Recommendation X.209 : 1988)

ISO/IEC 9646-1 : 1991, *Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 1: General concepts.*
(See also CCITT Recommendation X.290 : - [1])

ISO/IEC 9646-2 : 1991, *Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 2: Abstract test suite specification.*
(See also CCITT Recommendation X.291 : - [1])

ISO/IEC 9646-4 : 1991, *Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 4: Test realization.*
(See also CCITT Recommendation X.293 : - [1])

ISO/IEC 9646-5 : 1991, *Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 5: Requirements on test laboratories and clients for the conformance assessment process.* (See also CCITT Recommendation X.294 : - [1])

ISO/IEC 10646-1 : - [1], *Information technology - Multiple-Octet Coded Character Set - Part 1: Architecture and Basic Multilingual Plane.*

3 Definitions

3.1 Basic terms from ISO/IEC 9646-1

The following terms defined in ISO/IEC 9646-1 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

1) To be published.

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) test realizer
av) test step
aw) test suite
ax) test system
ay) upper tester
az) (test) verdict
ba) testing state

3.2 Terms from ISO 7498

The following terms defined in ISO 7498 apply:

a) application layer
b) protocol data unit
c) service access point
d) session layer
e) subnetwork

f) transfer syntax

g) transport layer

3.3 Terms from ISO/TR 8509

The following terms defined in ISO/TR 8509 apply:

a) service-provider

3.4 Terms from ISO/IEC 8824

The following terms defined in ISO/IEC 8824 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 ISO/IEC 8825

The following term defined in ISO/IEC 8825 applies:

encoding

3.6 TTCN specific terms

For the purposes of this part of ISO/IEC 9646 the following definitions apply:

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

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

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

3.6.4 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.

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

3.6.6 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.

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

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

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

3.6.10 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.

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

3.6.12 declarations part: That part of a TTCN test suite concerned with the definition and/or declaration of all non-

predefined components that are used in the test suite.

3.6.13 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.

3.6.14 default group: A named set of default behaviours.

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

3.6.16 default identifier: A unique name for a Default.

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

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

3.6.19 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.

3.6.20 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.

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

3.6.22 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.

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

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

3.6.25 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.

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

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

3.6.28 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.

3.6.29 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.

3.6.30 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.

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

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

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

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

3.6.35 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.

3.6.36 single constraint table: Declaration of a constraint for a single ASP or PDU of a given type arranged in a single table.

3.6.37 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.

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

3.6.39 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.

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

3.6.41 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.

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

3.6.43 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.

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

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

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

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

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

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

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

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

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

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

3.6.54 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.

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

3.6.56 tree identifier: A name identifying a local tree.

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

3.6.58 tree node: A single TTCN statement.

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

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

3.6.61 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.

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

4 Abbreviations

4.1 Abbreviations defined in ISO/IEC 9646-1.

For the purposes of this part of ISO/IEC 9646, the following abbreviations defined in ISO/IEC 9646-1:1991, clause 4 apply:

ATS : abstract test suite
ASP : abstract service primitive
ETS : executable test suite
IUT : implementation under test
LT : lower tester
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

4.2 Abbreviations defined in ISO/IEC 9646-2

For the purposes of this part of ISO/IEC 9646, the following abbreviations defined in ISO/IEC 9646-2:1991, clause 4 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 part of ISO/IEC 9646, the following abbreviations also apply:

ASN.1 : abstract syntax notation one
BNF : the extended Backus-Naur form used in TTCN
FDT : formal description technique
FIFO : first in first out
OSI : open systems interconnection
PDU : protocol data unit
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 The syntax forms of TTCN

TTCN is provided in two forms

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

b) 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 part of ISO/IEC 9646; 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 clause 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 and static semantics as described by the text and as described by annex A, then

a) except for the differences specified in clause A.5, the TTCN.MP syntax productions shall have precedence over the text and syntax productions in the body of this part of ISO/IEC 9646;

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

c) the static semantics restrictions specified in annex A shall have precedence over the text in the body of this part of ISO/IEC 9646.

If an ATS is specified in TTCN.GR in compliance with this part of ISO/IEC 9646, 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 part of ISO/IEC 9646 shall satisfy the requirements for either TTCN.GR or TTCN.MP.

NOTE - See ISO/IEC 9646-1:1991, clause 10, for an explanation of the use of the term "compliance" in ISO/IEC 9646.

6.2 ATSs that comply with the requirements of TTCN.GR shall satisfy the TTCN.GR syntax requirements stated in clauses 8 through 15 and clause A.4.

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

6.4 ATSs that comply with this part of ISO/IEC 9646 shall satisfy the static semantic requirements specified in clauses 7 through 15 and have operational semantics in accordance with the definition of the operational semantics in annex B, such that they are semantically valid.

6.5 A standardized ATS that complies with this part of ISO/IEC 9646 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) comply with ISO/IEC 9646-4.

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 ISO/IEC 9646-5 will record an abstract or executable test case error, depending on where the error is located.

6.6 Any standardized ATS that reached Draft International Standard status during or before 1991 or that is approved as a CCITT Recommendation during the 1989-1992 Study Period may be stated to comply with ISO/IEC 9646-3 but use some or all of the Draft International Standard (DIS) TTCN features which have changed between DIS and International Standard (IS), as outlined in annex F. Such a test suite shall reference ISO/IEC 9646-3 and contain a description of the differences between the features of TTCN that it uses and those specified in this IS specification of TTCN.

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 - The TTCN.MP Syntactic Metanotation

::=	is defined to be
|	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 substitute 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 tables:

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 below:

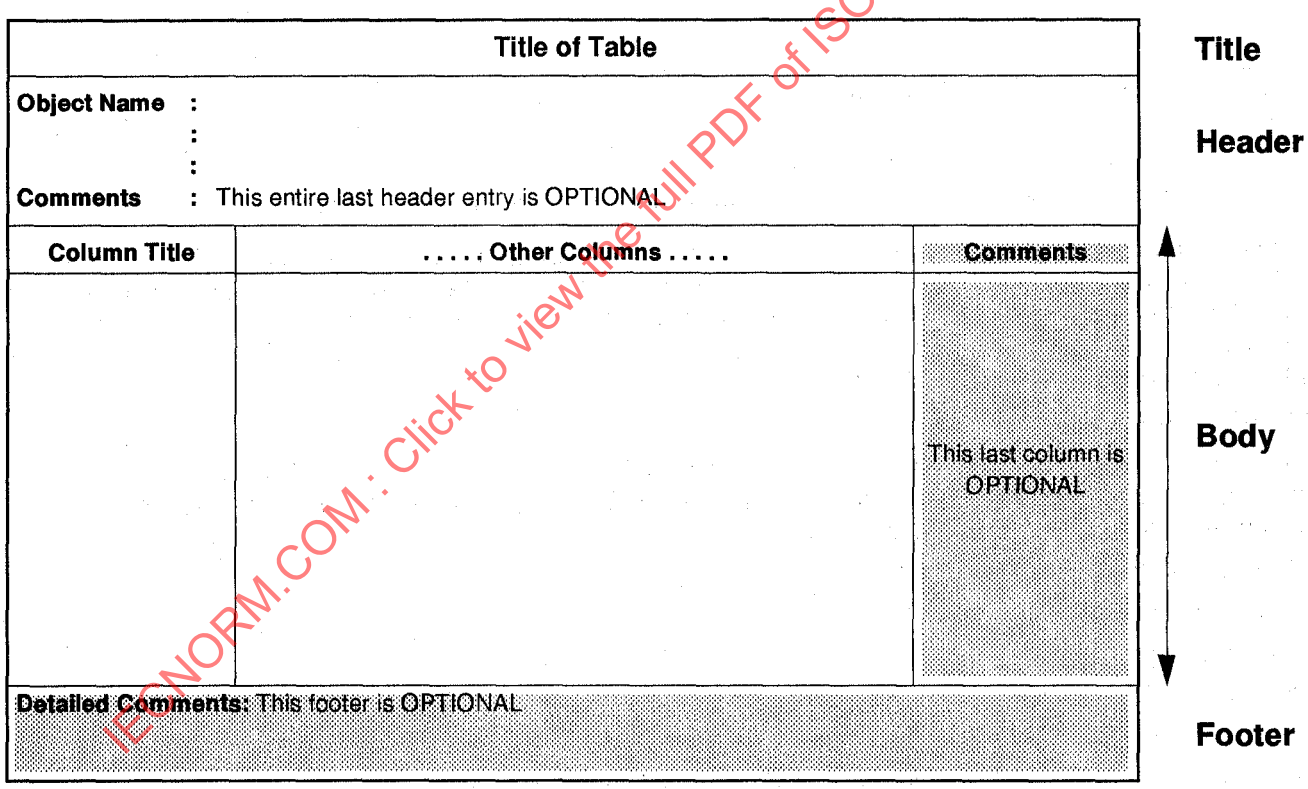


Figure 1 - Generalized layout of a single declaration table

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. The last item, named *Comments* contains an informal description of the object. This item may be omitted.

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 below:

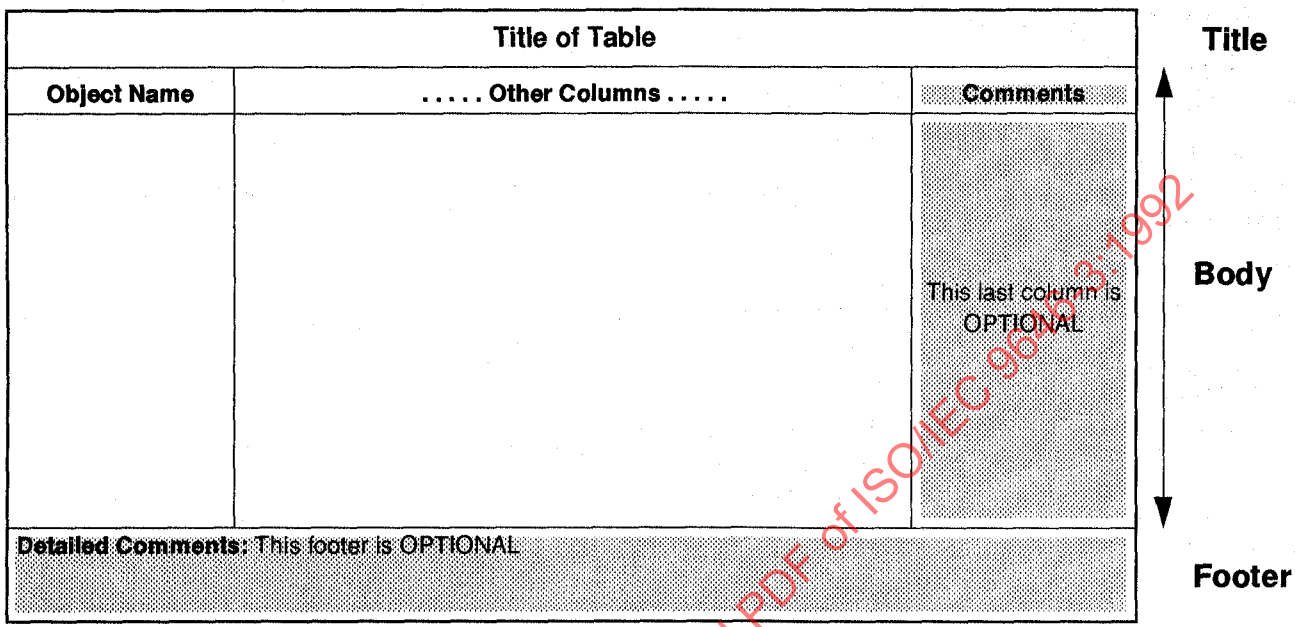


Figure 2 - Generalized layout of a multiple declaration table

This type of table has no header section. 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.

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 C.

7.3.5 Specification of proformas

This part of ISO/IEC 9646 specifies 32 types of TTCN.GR tables and provides a graphic view of the corresponding proformas. These proformas conform to the generalised layout 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 10646. 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 TTCN test suite structure

8.1 Introduction

TTCN allows a test suite to be hierarchically structured in accordance with ISO/IEC 9646-1:1991, 8.1. 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.

8.2 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 ISO/IEC 9646-1:1991, clause 8, figure 9). The Test Group References define the structure of the test suite. Test Group References shall have the following syntax:

SYNTAX DEFINITION:

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

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

8.3 Test Step Group References

8.3.1 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.

8.3.2 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:

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

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

8.4 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:

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

EXAMPLE 5 - Transport Default Group Reference: TRANSPORT/DEFAULT_LIBRARY/CLASS0/

8.5 Components 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 9),

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) Declarations Part (see clause 10),

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);

c) Constraints Part (see clause 11, 12, 13),

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.

d) Dynamic Part (see clause 14),

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).

9 Test Suite Overview

9.1 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 9.2);

b) Test Case Index (see 9.3);

c) Test Step Index (see 9.4);

d) Default Index (see 9.5).

9.2 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 ISO/IEC 9646-2:1991, clause 15);

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 how it has been derived; this 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;

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 the following proforma:

Test Suite Structure

Suite Name : *SuiteIdentifier*
Standards Ref : *FreeText*
PICS Ref : *FreeText*
PIXIT Ref : *FreeText*
Test Method(s) : *FreeText*
Comments : *[FreeText]*

Test Group Reference	Selection Ref	Test Group Objective	Page Nr
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
TestGroupReference	*[SelectExpr-Identifier]*	*FreeText*	*Number*
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.

Detailed Comments: *[FreeText]*

Proforma 1 - Test Suite Structure

SYNTAX DEFINITION:

3 SuiteIdentifier ::= Identifier
235 TestGroupReference ::= [SuiteIdentifier "/"] {TestGroupIdentifier "/"}
83 SelectExprIdentifier ::= Identifier

9.3 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 the following proforma:

Test Case Index				
Test Group Reference	**Test Case Id**	**Selection Ref**	**Description**	**Page Nr**
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
[TestGroupReference]	*TestCase Identifier*	*[SelectExpr-Identifier]*	*FreeText*	*Number*
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
Detailed Comments: *FreeText*				

Proforma 2 - Test Case Index

SYNTAX DEFINITION:

235 TestGroupReference ::= [SuiteIdentifier "/"] {TestGroupIdentifier "/"}
233 TestCaseIdentifier ::= Identifier
83 SelectExprIdentifier ::= Identifier

9.4 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 the following proforma:

Test Step Index			
Test Step Group Reference	**Test StepId**	**Description**	**Page Nr**
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
[TestStepGroupReference]	*TestStep Identifier*	*FreeText*	*Number*
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
Detailed Comments: *[FreeText]*			

Proforma 3 - Test Step Index

SYNTAX DEFINITION:

248 TestStepGroupReference ::= [SuiteIdentifier "/"] {TestStepGroupIdentifier "/"}
246 TestStepIdentifier ::= Identifier

9.5 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 the following proforma:

Default Index			
Default Group Reference	**DefaultId**	**Description**	**Page Nr**
[DefaultGroupReference]	*Default Identifier*	*FreeText*	*Number*
Detailed Comments: *[FreeText]*			

Proforma 4 - Default Index

SYNTAX DEFINITION:

258 DefaultGroupReference ::= [SuiteIdentifier "/"] {DefaultGroupIdentifier "/"}
257 DefaultIdentifier ::= Identifier

10 Declarations Part

10.1 Introduction

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

a) definitions:

1) Test Suite Types (see 10.2.3);

2) Test suite operations (see 10.3.4);

b) parameterization and selection of Test Cases:

1) Test Suite Parameters (see 10.4);

2) Test Case Selection Expressions (see 10.5);

c) declarations/definitions:

1) Test Suite Constants (see 10.6);

2) Test Suite Variables (see 10.7.1);

3) Test Case Variables (see 10.7.3);

4) PCO s (see 10.8);

5) Timers (see 10.9);

6) ASP types (see 10.10);

7) PDU types (see 10.11);

8) Aliases (see 10.15).

10.2 TTCN types

10.2.1 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.

10.2.2 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 6 - '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 7 - '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 8 - 'FF96'O

f) **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 of ISO/IEC 8824:1990.

Table 2 - Predefined CharacterString Types

NumericString
PrintableString
TeletexString (*i.e.*, T61 String)
VideotexString
VisibleString (*i.e.*, ISO646 String)
IA5String
GraphicString
GeneralString

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:

```
332  PredefinedType ::= INTEGER | BOOLEAN | BITSTRING | HEXSTRING | OCTETSTRING | CharacterString
333  CharacterString ::= NumericString | PrintableString | TeletexString | VideotexString | VisibleString | IA5String |
     GraphicString | GeneralString
338  Number ::= (NonZeroNum {Num} ) | 0
339  NonZeroNum ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
340  Num ::= 0 | NonZeroNum
341  BooleanValue ::= TRUE | FALSE
342  Bstring ::= "'" {Bin | Wildcard} "'" B
343  Bin ::= 0 | 1
344  Hstring ::= "'" {Hex | Wildcard} "'" H
345  Hex ::= Num | A | B | C | D | E | F
346  Ostring ::= "'" {Oct | Wildcard} "'" O
347  Oct ::= Hex Hex
348  Cstring ::= """ {Char | Wildcard | "\"} """
349  Char ::= /* REFERENCE - A character defined by the relevant character string type */
350  Wildcard ::= AnyOne | AnyOrNone
351  AnyOne ::= "?"
352  AnyOrNone ::= "*"
```

10.2.3 Test Suite Type Definitions

10.2.3.1 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).

10.2.3.2 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 4; only non-negative INTEGER literals or the keyword INFINITY for the upper bound shall be used.

This information shall be provided in the format shown in the following proforma:

Simple Type Definitions		
Type Name	**Type Definition**	**Comments**
SimpleTypeIdentifier	*Type&Restriction*	*[FreeText]*
Detailed Comments: *[FreeText]*		

Proforma 5 - Simple Type Definitions

SYNTAX DEFINITION:

```
27  SimpleTypeIdentifier ::= Identifier
29  Type&Restriction ::= Type [Restriction]
331 Type ::= PredefinedType | ReferenceType
332 PredefinedType ::= INTEGER | BOOLEAN | BITSTRING | HEXSTRING | OCTETSTRING | CharacterString
333 CharacterString ::= NumericString | PrintableString | TeletexString | VideotexString | VisibleString | IA5String |
    GraphicString | GeneralString
334 ReferenceType ::= TS_TypeIdentifier | ASP_Identifier | PDU_Identifier
335 TS_TypeIdentifier ::= SimpleTypeIdentifier | StructIdentifier | ASN1_TypeIdentifier
30  Restriction ::= LengthRestriction | IntegerRange | SimpleValueList
31  LengthRestriction ::= SingleTypeLength | RangeTypeLength
32  SingleTypeLength ::= "["Number "]"
33  RangeTypeLength ::= "[" LowerTypeBound To UpperTypebound "]"
34  IntegerRange ::= "(" LowerTypeBound To UpperTypebound ")"
35  LowerTypeBound ::= [Minus] Number | Minus INFINITY
36  UpperTypebound ::= [Minus] Number | INFINITY
37  To ::= TO | ".."
38  SimpleValueList::= "(" [Minus] LiteralValue {Comma [Minus] LiteralValue} ")"
```

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.

EXAMPLE 9 - Simple Test Suite Type definitions

Simple Type Definitions		
Type Name	**Type Definition**	**Comments**
Transport_classes	INTEGER(0, 1, 2, 3, 4)	classes that may be used for transport layer connection
String5	IA5String[5]	string of length 5
SeqNumbers	INTEGER(0..127)	all numbers from 0 to 127
PositiveNumbers	INTEGER(1..INFINITY)	all positive INTEGER numbers
String10to20	IA5String [10 .. 20]	string, min. length 10 characters and max. length 20 characters

10.2.3.3 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 standard, shall be used; if an abbreviation is used, then the full name shall follow in parentheses;

b) 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 standard, 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 10.12).

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 the following proforma:

Structured Type Definition		
Type Name : *StructId&FullId* **Comments :** *[FreeText]*		
Element Name	**Type Definition**	**Comments**
. . *ElemId&FullId* . .	. . *Type&Attributes* . .	. . *[FreeText]* . .
Detailed Comments: *[FreeText]*		

Proforma 6 - Structured Type Definition

SYNTAX DEFINITION:

42 StructId&FullId ::= StructIdentifier [FullIdentifier]
44 StructIdentifier ::= Identifier
48 ElemId&FullId ::= ElemIdentifier [FullIdentifier]
49 ElemIdentifier ::= Identifier
43 FullIdentifier ::= "(" BoundedFreeText ")"
154 Type&Attributes::= (Type [LengthAttribute]) | **PDU**
331 Type ::= PredefinedType | ReferenceType
332 PredefinedType ::= **INTEGER** | **BOOLEAN** | **BITSTRING** | **HEXSTRING** | **OCTETSTRING** | CharacterString
333 CharacterString ::= **NumericString** | **PrintableString** | **TeletexString** | **VideotexString** | **VisibleString** | **IA5String** | **GraphicString** | **GeneralString**
334 ReferenceType ::= TS_TypeIdentifier | ASP_Identifier | PDU_Identifier
335 TS_TypeIdentifier ::= SimpleTypeIdentifier | StructIdentifier | ASN1_TypeIdentifier
155 LengthAttribute ::= SingleLength | RangeLength
156 SingleLength ::= "[" Bound "]"
157 Bound ::= Number | TS_ParIdentifier | TS_ConstIdentifier
158 RangeLength ::= "[" LowerBound To UpperBound "]"
159 LowerBound ::= Bound
37 To ::= **TO** | ".."
160 UpperBound ::= Bound | **INFINITY**

10.2.3.4 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 ISO/IEC 8824. The following information shall be supplied for each ASN.1type:

a) its name,

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

b) the ASN.1type definition,

which shall follow the syntax defined in ISO/IEC 8824. For identifiers within that definition the dash 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 ISO/IEC 8824 ASN.1 comments can be used within the table body. The comments column shall not be present in this table.

This information shall be provided in the following proforma:

ASN.1 Type Definition
Type Name : *ASN1_TypeId&FullId* **Comments :** *[FreeText]*
Type Definition
ASN1_Type&LocalTypes
Detailed Comments: *[FreeText]*

Proforma 7 - ASN.1 Type Definition

SYNTAX DEFINITION:

54 ASN1_TypeId&FullId ::= ASN1_TypeIdentifier [FullIdentifier]
55 ASN1_TypeIdentifier ::= Identifier
43 FullIdentifier ::= "(" BoundedFreeText ")"
57 ASN1_Type&LocalTypes ::= ASN1_Type {ASN1_LocalType}
58 ASN1_Type ::= Type
/* REFERENCE -Where Type is a non-terminal defined in ISO/IEC 8824 */
59 ASN1_LocalType ::= Typeassignment
/* REFERENCE -Where Typeassignment is a non-terminal defined in ISO/IEC 8824 */

EXAMPLE 10 - 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(1), last(31)}
 	
-- MONTH_type and YEAR_type are defined in other ASN.1 Type Definitions tables

10.2.3.5 ASN.1 Type Definitions by Reference

Types can be specified by a precise reference to an ASN.1 type defined in an OSI standard 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 ISO/IEC 8824;

c) the module identifier,

which consists of a module reference that shall follow the identifier rules stated in ISO/IEC 8824, and an optional ObjectIdentifier; the module shall be unique within the domain of interest.

This information shall be provided in the following proforma:

ASN.1 Type Definitions By Reference			
Type Name	**Type Reference**	**Module Identifier**	**Comments**
. . *ASN1_TypeId&FullId* . .	. . *TypeReference* . .	. . *ModuleIdentifier* . .	. . *[FreeText]* . .
Detailed Comments: *[FreeText]*			

Proforma 8 - ASN.1 Type Definitions By Reference

SYNTAX DEFINITION:

```
54  ASN1_TypeId&FullId ::= ASN1_TypeIdentifier [FullIdentifier]
55  ASN1_TypeIdentifier ::= Identifier
43  FullIdentifier ::= "(" BoundedFreeText ")"
63  TypeReference ::= typereference
    /* REFERENCE -Where typereference is defined in ISO/IEC 8824:1990, subclause 8.2 */
65  ModuleIdentifier ::= ModuleIdentifier
    /* REFERENCE -Where ModuleIdentifier is a non-terminal defined in ISO/IEC 8824 */
```

Since the ASN.1 types imported from ASN.1 modules can contain identifiers, type references and value references that follow the identifier rules in ISO/IEC 8824, 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 11 - 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	{	field1 field2	Sub_Type_1, BIT STRING {first_bit(0), second_bit(1) } }

10.3 TTCN operators and TTCN operations

10.3.1 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.

10.3.2 TTCN operators

10.3.2.1 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.

Table 3 - Precedence of Operators

highest ↓ lowest	Unary	( )
		+ - NOT
		* / MOD AND
	Binary	+ - OR
		= < > <> >= <=

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.

SYNTAX DEFINITION:

```
321  AddOp ::= "+" | "-" | OR
322  MultiplyOp ::= "*" | "/" | MOD | AND
323  UnaryOp ::= "+" | "-" | NOT
324  RelOp::= "=" | "<" | ">" | "<>" | ">=" | "<="
```

10.3.2.2 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.

10.3.2.3 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 11.6.5.

10.3.2.4 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.

10.3.3 Predefined operations

10.3.3.1 Introduction

The predefined operations fall into two categories:

a) conversion;

b) others

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

10.3.3.2 Predefined conversion operations

10.3.3.2.1 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.

10.3.3.2.2 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.

10.3.3.2.3 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.

10.3.3.2.4 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.

10.3.3.2.5 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.

10.3.3.3 Other predefined operations

TTCN also defines the following predefined operations:

a) IS_PRESENT;

b) NUMBER_OF_ELEMENTS;

c) IS_CHOSEN;

d) LENGTH_OF;

10.3.3.3.1 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 10.3.4.

EXAMPLE 12 - 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.
```

10.3.3.3.2 NUMBER_OF_ELEMENTS(DataObjectReference) ⇒ INTEGER

The operation returns the actual number of elements of a data object that is of type ASN.1 SEQUENCE OF or SET OF. The operation shall not be applied to data objects or fields of data objects other than of ASN.1 type SEQUENCE OF or SET OF. The argument of the operation shall have the format as defined in 10.3.4.

EXAMPLE 13 - 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.
```

10.3.3.3.3 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 the of ASN.1 type CHOICE. The argument of the operation shall have the format as defined in 10.3.4.

EXAMPLE 14 - 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.

10.3.3.3.4 LENGTH_OF(DataObjectReference) ⇒ INTEGER

The operation returns the actual length of a data object that is of type BITSTRING, HEXSTRING, OCTETSTRING or CharacterString. The units of length for each string type are defined in table 4. The argument of the operation shall have the format as defined in 10.3.4.

The operation shall not be applied to data objects or fields of data objects other than of BITSTRING, HEXSTRING, OCTETSTRING or CharacterString.

EXAMPLE 15 - Use of LENGTH_OF

If S='010'B then LENGTH_OF(S) returns 3

If S='F3'H then LENGTH_OF(S) returns 2

If S='F2'O then LENGTH_OF(S) returns 1

If S="EXAMPLE" then LENGTH_OF(S) returns 7

10.3.4 Test Suite Operation Definitions

Operations specific to a test suite may be introduced by the ATS specifier. To define a new operation, the following information 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's type.

When more than one parameter of the same type is used, the parameters may be specified as a parameter sub-list. When a parameter sub-list 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.

EXAMPLE 16 - 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 description of the operation,

which shall consist of an explanation of the operation, plus at least one example showing an invocation and corresponding result; the explanation shall 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.

This information shall be provided in the format shown in the following proforma:

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

Proforma 9 - Test Suite Operation Definition

SYNTAX DEFINITION:

69 TS_OpId&ParList ::= TS_OpIdentifier [FormalParList]
331 Type ::= PredefinedType | ReferenceType
332 PredefinedType ::= **INTEGER** | **BOOLEAN** | **BITSTRING** | **HEXSTRING** | **OCTETSTRING** | CharacterString
333 CharacterString ::= **NumericString** | **PrintableString** | **TeletexString** | **VideotexString** | **VisibleString** | **IA5String** | **GraphicString** | **GeneralString**
334 ReferenceType ::= TS_TypeIdentifier | ASP_Identifier | PDU_Identifier
335 TS_TypeIdentifier ::= SimpleTypeIdentifier | StructIdentifier | ASN1_TypeIdentifier

An operation may be compared to a function in an ordinary programming language. However, the parameters to the operation shall not be altered as a result of any call of the operation and there shall be no side effects (*e.g.,* no changes to any Test Suite or Test Case Variable).

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; 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; and

EXAMPLE 17 - Definition of the operation SUBSTR:

Test Suite Operation Definition
Operation Name : SUBSTR (source:IA5String; start_index, length:INTEGER) **Result Type** : IA5String **Comments** :
Description
SUBSTR(source, start_index, length) is the string of length *length* starting from index *start_index* of the source string *source.* For example: SUBSTR("abcde",3,2) = "cd" SUBSTR("abcde",1,3) = "abc"

10.4 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) 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 the following proforma:

Test Suite Parameter Declarations			
Parameter Name	**Type**	**PICS/PIXIT Ref**	**Comments**
. . *TS_ParIdentifier* . .	. . *Type* . .	. . *FreeText* .	. . *[FreeText]* . .
Detailed Comments: *[FreeText]*			

Proforma 10 - Test Suite Parameter Declarations

SYNTAX DEFINITION:

77 TS_ParIdentifier ::= Identifier
331 Type ::= PredefinedType | ReferenceType
332 PredefinedType ::= **INTEGER** | **BOOLEAN** | **BITSTRING** | **HEXSTRING** | **OCTETSTRING** | CharacterString
333 CharacterString ::= **NumericString** | **PrintableString** | **TeletexString** | **VideotexString** | **VisibleString** | **IA5String** | **GraphicString** | **GeneralString**
334 ReferenceType ::= TS_TypeIdentifier | ASP_Identifier | PDU_Identifier
335 TS_TypeIdentifier ::= SimpleTypeIdentifier | StructIdentifier | ASN1_TypeIdentifier

EXAMPLE 18 - Declaration of Test Suite Parameters:

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

10.5 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 ISO/IEC 9646-2.

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 selectionexpression 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 the following proforma:

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

Proforma 11 - Test Case Selection Expression Definitions

SYNTAX DEFINITION:

83 SelectExprIdentifier ::= Identifier
85 SelectionExpression ::= Expression

10.6 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, an ASN.1 type, a Test Suite Type or a PDU type;

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 the following proforma:

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

Proforma 12 - Test Suite Constant Declarations

SYNTAX DEFINITION:

90 TS_ConstIdentifier ::= Identifier
331 Type ::= PredefinedType | ReferenceType

332 PredefinedType ::= **INTEGER** | **BOOLEAN** | **BITSTRING** | **HEXSTRING** | **OCTETSTRING** | CharacterString
333 CharacterString ::= **NumericString** | **PrintableString** | **TeletexString** | **VideotexString** | **VisibleString** | **IA5String** | **GraphicString** | **GeneralString**
334 ReferenceType ::= TS_TypeIdentifier | ASP_Identifier | PDU_Identifier
335 TS_TypeIdentifier ::= SimpleTypeIdentifier | StructIdentifier | ASN1_TypeIdentifier
93 DeclarationValue ::= Expression

EXAMPLE 19 - Declaration of Test Suite Constants

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

10.7 TTCN variables

10.7.1 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.

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 the following proforma:

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

Proforma 13 - Test Suite Variable Declarations

SYNTAX DEFINITION:

97 TS_VarIdentifier ::= Identifier
331 Type ::= PredefinedType | ReferenceType
332 PredefinedType ::= **INTEGER** | **BOOLEAN** | **BITSTRING** | **HEXSTRING** | **OCTETSTRING** | CharacterString
333 CharacterString ::= **NumericString** | **PrintableString** | **TeletexString** | **VideotexString** | **VisibleString** | **IA5String** | **GraphicString** | **GeneralString**
334 ReferenceType ::= TS_TypeIdentifier | ASP_Identifier | PDU_Identifier
335 TS_TypeIdentifier ::= SimpleTypeIdentifier | StructIdentifier | ASN1_TypeIdentifier
93 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 20 - 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.

10.7.2 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 14.10.4);

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.

10.7.3 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. 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 the following proforma:

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

Proforma 14 - Test Case Variable Declarations

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.

SYNTAX DEFINITION:

103 TC_VarIdentifier ::= Identifier
331 Type ::= PredefinedType | ReferenceType
332 PredefinedType ::= **INTEGER** | **BOOLEAN** | **BITSTRING** | **HEXSTRING** | **OCTETSTRING** | CharacterString
333 CharacterString ::= **NumericString** | **PrintableString** | **TeletexString** | **VideotexString** | **VisibleString** | **IA5String** | **GraphicString** | **GeneralString**
334 ReferenceType ::= TS_TypeIdentifier | ASP_Identifier | PDU_Identifier
335 TS_TypeIdentifier ::= SimpleTypeIdentifier | StructIdentifier | ASN1_TypeIdentifier
93 DeclarationValue ::= Expression

10.7.4 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 Variable appears on the left-hand side of an assignment statement (see 14.10.4);

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.

10.8 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.

The number of PCOs shall be as defined in ISO/IEC 9646-1: 1991, 7.5, and ISO/IEC 9646-2: 1991, 12.6, for the test method(s) identified in the Test Suite Structure table.

TTCN behaviour statements specified for execution at the UT PCO shall not place requirements beyond those specified by ISO/IEC 9646-2.

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 is successful by being passed from the LT to the service-provider, or by being 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 - 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;

c) its role,

which is an explanation of which type of tester is placed at the PCO. The predefined identifier **UT** indicates that the PCO is an upper tester PCO and **LT** specifies a lower tester PCO.

This information shall be provided in the format shown in the following proforma:

PCO Declarations			
PCO Name	**PCO Type**	**Role**	**Comments**
. . *PCO_Identifier* . .	. . *PCO_TypeIdentifier* . .	. . *PCO_Role* . .	. . *[FreeText]* . .
Detailed Comments: *[FreeText]*			

Proforma 15 - PCO Declarations

SYNTAX DEFINITION:

109 PCO_Identifier ::= Identifier
111 PCO_TypeIdentifier ::= Identifier
113 PCO_Role ::= **UT** | **LT**

EXAMPLE 21 - Declaration of PCOs

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

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 SAPs (Service Access Point) 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 - 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).

10.9 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) **us** (*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 the following proforma:

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

Proforma 16 - Timer Declarations

SYNTAX DEFINITION:

117 TimerIdentifier ::= Identifier
93 DeclarationValue ::= Expression
120 TimeUnit ::= **ps** | **ns** | **us** | **ms** | **s** | **min**

EXAMPLE 22 - Declaration of timers

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

10.10 ASP Type Definitions

10.10.1 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.

10.10.2 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 standard, 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 standard, 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 10.12. The length values shall be interpreted according to table 4. 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 the following proforma:

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

Proforma 17 - ASP Type Definition

SYNTAX DEFINITION:

127 ASP_Id&FullId ::= ASP_Identifier [FullIdentifier]

128 ASP_Identifier ::= Identifier
43 FullIdentifier ::= "(" BoundedFreeText ")"
111 PCO_TypeIdentifier ::= Identifier
132 ASP_ParIdOrMacro ::= ASP_ParId&FullId | MacroSymbol
133 ASP_ParId&FullId ::= ASP_ParIdentifier [FullIdentifier]
134 ASP_ParIdentifier ::= Identifier
150 MacroSymbol ::= "<-"
154 Type&Attributes::= (Type [LengthAttribute]) | **PDU**
331 Type ::= PredefinedType | ReferenceType
332 PredefinedType ::= **INTEGER** | **BOOLEAN** | **BITSTRING** | **HEXSTRING** | **OCTETSTRING** | CharacterString
333 CharacterString ::= **NumericString** | **PrintableString** | **TeletexString** | **VideotexString** | **VisibleString** | **IA5String** | **GraphicString** | **GeneralString**
334 ReferenceType ::= TS_TypeIdentifier | ASP_Identifier | PDU_Identifier
335 TS_TypeIdentifier ::= SimpleTypeIdentifier | StructIdentifier | ASN1_TypeIdentifier
155 LengthAttribute ::= SingleLength | RangeLength
156 SingleLength ::= "[" Bound "]"
157 Bound ::= Number | TS_ParIdentifier | TS_ConstIdentifier
158 RangeLength ::= "[" LowerBound To UpperBound "]"
159 LowerBound ::= Bound
37 To ::= **TO** | ".."
160 UpperBound ::= Bound | **INFINITY**

EXAMPLE 23 - T_CONNECTrequest Abstract Service Primitive

The figure below shows an example from the Transport Service [ISO 8072]. 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 [ISO 8073].

ASP Type Definition		
ASP Name : CONreq (T_CONNECTrequest) **PCO Type :** TSAP **Comments :**		
Parameter Name	**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		

10.10.3 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.

10.10.4 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 ISO/IEC 8824. The following information shall be supplied for each ASN.1 ASP:

a) its name,

where the full name, as given in the appropriate protocol standard, 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 ISO/IEC 8824. 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.

This information shall be provided in the following proforma:

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

Proforma 18 - ASN.1 ASP Type Definition

SYNTAX DEFINITION:

```
127  ASP_Id&FullId ::= ASP_Identifier [FullIdentifier]
128  ASP_Identifier ::= Identifier
43   FullIdentifier ::= "(" BoundedFreeText ")"
111  PCO_TypeIdentifier ::= Identifier
57   ASN1_Type&LocalTypes ::= ASN1_Type {ASN1_LocalType}
58   ASN1_Type ::= Type
     /* REFERENCE -Where Type is a non-terminal defined in ISO/IEC 8824 */
59   ASN1_LocalType ::= Typeassignment
     /* REFERENCE -Where Typeassignment is a non-terminal defined in ISO/IEC 8824 */
```

10.10.5 ASN.1 ASP Type Definitions by Reference

ASPs can be specified by a precise reference to an ASN.1 ASP defined in an OSI standard 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 ISO/IEC 8824;

d) the module identifier,

which consists of a module reference that shall follow the identifier rules stated in ISO/IEC 8824 and an optional ObjectIdentifier.

This information shall be provided in the following proforma:

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

Proforma 19 - ASN.1 ASP Type Definitions By Reference

SYNTAX DEFINITION:

127 ASP_Id&FullId ::= ASP_Identifier [FullIdentifier]
128 ASP_Identifier ::= Identifier
43 FullIdentifier ::= "(" BoundedFreeText ")"
111 PCO_TypeIdentifier ::= Identifier
63 TypeReference ::= typereference
/* REFERENCE -Where typereference is defined in ISO/IEC 8824:1990, subclause 8.2 */
65 ModuleIdentifier ::= ModuleIdentifier
/* REFERENCE -Where ModuleIdentifier is a non-terminal defined in ISO/IEC 8824 */

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).

10.11 PDU Type Definitions

10.11.1 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 standard, 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.

10.11.2 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 standard, 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) 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 standard, 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 10.12. The length values shall be interpreted according to table 4. 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.

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 the following proforma:

PDU Type Definition		
PDU Name : *PDU_Id&FullId* **PCO Type :** *[PCO_TypeIdentifier]* **Comments :** *[FreeText]*		
Field Name	**Field Type**	**Comments**
. . *PDU_FieldIdOrMacro* . .	. . *Type&Attributes* . .	. . *[FreeText]* . .
Detailed Comments: *[FreeText]*		

Proforma 20 - PDU Type Definition

SYNTAX DEFINITION:

```
144  PDU_Id&FullId ::= PDU_Identifier [FullIdentifier]
145  PDU_Identifier ::= Identifier
43   FullIdentifier ::= "(" BoundedFreeText ")"
```

111 PCO_TypeIdentifier ::= Identifier
149 PDU_FieldIdOrMacro ::= PDU_FieldId&FullId | MacroSymbol
151 PDU_FieldId&FullId ::= PDU_FieldIdentifier [FullIdentifier]
152 PDU_FieldIdentifier ::= Identifier
150 MacroSymbol ::= "<-"
154 Type&Attributes::= (Type [LengthAttribute]) | **PDU**
331 Type ::= PredefinedType | ReferenceType
332 PredefinedType ::= **INTEGER** | **BOOLEAN** | **BITSTRING** | **HEXSTRING** | **OCTETSTRING** | CharacterString
333 CharacterString ::= **NumericString** | **PrintableString** | **TeletexString** | **VideotexString** | **VisibleString** | **IA5String** | **GraphicString** | **GeneralString**
334 ReferenceType ::= TS_TypeIdentifier | ASP_Identifier | PDU_Identifier
335 TS_TypeIdentifier ::= SimpleTypeIdentifier | StructIdentifier | ASN1_TypeIdentifier
155 LengthAttribute ::= SingleLength | RangeLength
156 SingleLength ::= "[" Bound "]"
157 Bound ::= Number | TS_ParIdentifier | TS_ConstIdentifier
158 RangeLength ::= "[" LowerBound To UpperBound "]"
159 LowerBound ::= Bound
37 To ::= **TO** | ".."
160 UpperBound ::= Bound | **INFINITY**

EXAMPLE 24 - A typical PDU Type Definition

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

10.11.3 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.

10.11.4 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 ISO/IEC 8824. The following information shall be supplied for each ASN.1 PDU:

a) its name,

where the full name, as given in the appropriate protocol standard, 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 ASN.1 PDU type definition,

which shall follow the syntax defined in ISO/IEC 8824. 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.

This information shall be provided in the following proforma:

ASN.1 PDU Type Definition
PDU Name : *PDU_Id&FullId* **PCO Type :** *[PCO_TypeIdentifier]* **Comments :** *[FreeText]*
Type Definition
ASN1_Type&LocalTypes
Detailed Comments: *[FreeText]*

Proforma 21 - ASN.1 PDU Type Definition

SYNTAX DEFINITION:

144 PDU_Id&FullId ::= PDU_Identifier [FullIdentifier]
145 PDU_Identifier ::= Identifier
43 FullIdentifier ::= "(" BoundedFreeText ")"
111 PCO_TypeIdentifier ::= Identifier
57 ASN1_Type&LocalTypes ::= ASN1_Type {ASN1_LocalType}
58 ASN1_Type ::= Type
/* REFERENCE -Where Type is a non-terminal defined in ISO/IEC 8824 */
59 ASN1_LocalType ::= Typeassignment
/* REFERENCE -Where Typeassignment is a non-terminal defined in ISO/IEC 8824 */

EXAMPLE 25 - 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,
-- etc.
}

10.11.5 ASN.1 PDU Type Definitions by Reference

PDUs can be specified by a precise reference to an ASN.1 PDU defined in an OSI standard 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 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 ISO/IEC 8824;

d) the module identifier,

which consists of a module reference that shall follow the identifier rules stated in ISO/IEC 8824 and an optional ObjectIdentifier.

This information shall be provided in the following proforma:

ASN.1 PDU Type Definitions By Reference				
PDU Name	**PCO Type**	**Type Reference**	**Module Identifier**	**Comments**
. . *PDU_Id&FullId* . .	. . *[PCO_TypeIdentifier]* . .	. . *TypeReference* . .	. . *ModuleIdentifier* . .	. . *[FreeText]* . .
Detailed Comments: *[FreeText]*				

Proforma 22 - ASN.1 PDU Type Definitions By Reference

SYNTAX DEFINITION:

```
144  PDU_Id&FullId ::= PDU_Identifier [FullIdentifier]
145  PDU_Identifier ::= Identifier
43   FullIdentifier ::= "(" BoundedFreeText ")"
111  PCO_TypeIdentifier ::= Identifier
63   TypeReference ::= typereference
     /* REFERENCE -Where typereference is defined in ISO/IEC 8824:1990, subclause 8.2 */
65   ModuleIdentifier ::= ModuleIdentifier
     /* REFERENCE -Where ModuleIdentifier is a non-terminal defined in ISO/IEC 8824 */
```

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).

10.12 String length specifications

10.12.1 TTCN permits the specification of length restrictions on string types (*i.e.*, BITSTRING, HEXSTRING, OCTETSTRING and all CharacterString types) 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.

10.12.2 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.

The following table specifies the units of length for different string types:

Table 4 - Units of length used in field length specifications

Type	Units of Length
BITSTRING	Bits
HEXSTRING	Hex digits
OCTETSTRING	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 26 - 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.

10.13 ASP and PDU 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 clause 11, 12, 13) 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.

10.14 ASP and PDU Definitions for RECEIVE events

For ASPs and/or PDUs received by the tester the ASP 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.

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.

10.15 Alias Definitions

10.15.1 Introduction

In order to enhance the readability of TTCN behaviour descriptions, an Alias may be used to h 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 the format shown in the following proforma:

Alias Definitions		
Alias Name	**Expansion**	**Comments**
AliasIdentifier	*Expansion*	*[FreeText]*
Detailed Comments: *[FreeText]*		

Proforma 23 - Alias Definitions

SYNTAX DEFINITION:

168 AliasIdentifier ::= Identifier
170 Expansion ::= ASP_Identifier | PDU_Identifier

10.15.2 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 27 - 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.

11 Constraints Part

11.1 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 14) 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 12);

b) ASN.1 constraints (see clause 13).

11.2 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. 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.

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 literal values, Test Suite Parameters, Test Suite Constants, formal parameters and Test Suite Operations.

To facilitate static chaining a (possibly parameterized) constraints reference is also allowed as a parameter or field value.

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 11.6.2.

11.3 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's type. If more than one parameter of the same type is used, the parameter may be specified as a parameter sub-list. When a parameter sub-

list is used, the parameter names shall be separated by a comma. The final parameter in the sub-list shall be followed by a colon and the name of the parameter sub-list's type. When more than one parameter and type pair (or parameter sub-list 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.

11.4 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 annex D for examples of static and dynamic chaining).

Wherever constraints are referenced within constraints declarations, those references shall not be recursive (neither directly or indirectly).

11.5 Constraints for SEND events

Constraints that are referenced for SEND events shall not include wildcards (*i.e.*, AnyValue (?) 0r 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 11.6.2 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.

11.6 Constraints for RECEIVE events

11.6.1 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 11.6.3, 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 ASP parameters and/or PDU fields are of the type specified in the ASP and/or PDU definitions; if the value, alphabet and length satisfies any restriction associated with the type; and if the ASP parameter and/or PDU field values correctly match those of the constraint.

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.

11.6.2 Matching mechanisms

An overview of the supported matching mechanisms is shown in table 5, 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 clauses.

The shaded area in table 5 indicates the mechanisms that apply to both predefined TTCN and ASN.1 types.

Table 5 - 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	•	•	•	•	•	•				•	•		•	•
CHARSTRINGS	•	•	•	•	•	•				•	•		•	•
SEQUENCE	•	•	•	•	•	•								•
SEQUENCE OF	•	•	•	•	•	•				•	•	•	•	•
SET	•	•	•	•	•	•								•
SET OF	•	•	•	•	•	•		•	•	•	•		•	•
ANY	•	•	•	•	•	•								•
CHOICE	•	•	•	•	•	•								•
OBJECT ID	•	•	•	•	•	•								•

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.

11.6.3 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.

11.6.4 Instead of Value

11.6.4.1 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:

201 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 28 - Constraints using Complement instead of a value, and with a value list:

Type	Constraint
INTEGER	COMPLEMENT(5)
INTEGER	COMPLEMENT(1, 3, 5)

11.6.4.2 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.

In tabular constraints Omit shall be denoted by dash (-). In ASN.1 constraints Omit is denoted by **OMIT**.

SYNTAX DEFINITION:

202 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 29 - Constraints using Omit instead of a value, at top level:

Type	Constraint
INTEGER OPTIONAL	OMIT

11.6.4.3 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:

203 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 30 - Constraints using Value in combination with AnyValue:

Type	Constraint
SEQUENCE OF SET OF INTEGER	{ {1, 2}, ?, {1, 2, ?} }

11.6.4.4 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 11.6.4.4 and 11.6.5.2.

SYNTAX DEFINITION:

204 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 31 - Constraints using Value in combination with AnyOrOmit:

Type	Constraint
SEQUENCE OF { id1 SET OF INTEGER	{ id1 {2, 5},
id2 SET OF INTEGER	id2 * }

11.6.4.5 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:

205 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 32 - Constraints using ValueList instead of a specific value, for INTEGER type:

Type	Constraint
INTEGER	(2, 4, 6)

EXAMPLE 33 - Constraints using ValueList instead of a specific value, for CHOICE type:

Type	Constraint
CHOICE { a INTEGER,	(a 2, b TRUE)
b BOOLEAN }	

11.6.4.6 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:

206 ValueRange ::= "(" ValRange ")"
207 ValRange ::= (LowerRangeBound To UpperRangeBound)
208 LowerRangeBound ::= ConstraintExpression | Minus **INFINITY**
209 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 34 - Constraints using Range instead of a value:

Type	Constraint
INTEGER	(1 .. 6)
	(-INFINITY .. 8)
	(12 .. INFINITY)

11.6.4.7 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:

210 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 35 - Constraints using SuperSet instead of a specific value:

Type	Constraint
SET OF INTEGER	SUPERSET({1, 2, 3})

11.6.4.8 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:

211 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 SuperSet mechanism is used.

EXAMPLE 36 - Constraints using SuperSet instead of a specific value:

Type	Constraint
SET OF INTEGER	SUBSET({2, 4, 6, 8, 10})

11.6.5 Inside Values

11.6.5.1 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:

351 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 37 - 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.

11.6.5.2 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:

352 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 38 - Constraints using AnyOne:

Type	Constraint
IA5String	"ab*z"
SEQUENCE OF INTEGER	{1, 2, *, 10 }
SEQUENCE OF IA5String	{ "ab*z", *, "abc" }

11.6.5.3 Permutation

Permutation 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:

212 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 39 - Constraints using Permutation:

Type	Constraint
SEQUENCE OF INTEGER	{PERMUTATION (1, 2, 3), 5}

EXAMPLE 40 - 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}.

11.6.6 Attributes of values

11.6.6.1 Length

Length is an operation for matching that can be used only as an attribute of the following mechanisms: Specific value, Complement, Omit, AnyValue, AnyOrOmit, AnyOne, AnyOrNone and Permutation.

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 10.12. The units of length are to interpreted according to table 4. 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:

214 ValueLength ::= SingleValueLength | RangeValueLength
215 SingleValueLength ::= "[" ValueBound "]"
216 ValueBound ::= Number | TS_ParIdentifier | TS_ConstIdentifier | FormalParIdentifier
217 RangeValueLength ::= "[" LowerValueBound To UpperValueBound "]"

218 LowerValueBound ::= ValueBound
37 To ::= **TO** | ".."
219 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.

EXAMPLE 41 - Constraints using Value in combination with Length:

Type	Constraint
IA5String	"ab*ab" [13]

11.6.6.2 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 42 - Constraints using Value in combination with IfPresent:

Type	Constraint
IA5String OPTIONAL	"abcdef" IF_PRESENT

12 Specification of constraints using tables

12.1 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.

12.2 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. Structure constraint tables are very similar to PDU constraint tables. The element values for structure constraints shall be provided in the format shown in the following proforma:

Structured Type Constraint Declaration		
Constraint Name : *ConsId&ParList* **Structured Type** : *StructIdentifier* **Derivation Path** : *[DerivationPath]* **Comments** : *[FreeText]*		
Element Name	**Element Value**	**Comments**
ElemIdentifier	*ConstraintValue&Attributes*	*[FreeText]*
Detailed Comments: *[FreeText]*		

Proforma 24 - Structured Type Constraint Declaration

This proforma is used in the same way that the PDU Constraint Declaration proforma is used for PDUs (see 12.4).

SYNTAX DEFINITION:

190 ConsId&ParList ::= ConstraintIdentifier [FormalParList]
191 ConstraintIdentifier ::= Identifier
44 StructIdentifier ::= Identifier
193 DerivationPath ::= {ConstraintIdentifier Dot}+
49 ElemIdentifier ::= Identifier
197 ConstraintValue&Attributes ::= ConstraintValue ValueAttributes
198 ConstraintValue::= ConstraintExpression | MatchingSymbol | ConsRef
199 ConstraintExpression ::= Expression
200 MatchingSymbol ::= Complement | Omit | AnyValue | AnyOrOmit | ValueList | ValueRange | SuperSet | SubSet | Permutation
277 ConsRef ::= ConstraintIdentifier [ActualCrefParList]
278 ActualCrefParList ::= "(" ActualCrefPar {Comma ActualCrefPar} ")"
279 ActualCrefPar ::= Value | DataObjectIdentifier | ConsRef
213 ValueAttributes ::= [ValueLength] **[IF_PRESENT]**
214 ValueLength ::= SingleValueLength | RangeValueLength
215 SingleValueLength ::= "[" ValueBound "]"
216 ValueBound ::= Number | TS_ParIdentifier | TS_ConstIdentifier | FormalParIdentifier
217 RangeValueLength ::= "[" LowerValueBound To UpperValueBound "]"
218 LowerValueBound ::= ValueBound
37 To ::= **TO** | ".."
219 UpperValueBound ::= ValueBound | **INFINITY**

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.

12.3 ASP Constraint Declarations

The parameter values for ASP constraints shall be provided in the format shown in the following proforma:

ASP Constraint Declaration		
Constraint Name : *ConsId&ParList* **ASP Type** : *ASP_Identifier* **Derivation Path** : *[DerivationPath]* **Comments** : *[FreeText]*		
Parameter Name	**Parameter Value**	**Comments**
. . *ASP_ParIdOrMacro* . .	. . *ConstraintValue&Attributes* . .	. . *[FreeText]* . .
Detailed Comments: *[FreeText]*		

Proforma 25 - ASP Constraint Declaration

This proforma is used for ASPs in the same way that the PDU Constraint Declaration proforma is used (see 12.4).

SYNTAX DEFINITION:

190 ConsId&ParList ::= ConstraintIdentifier [FormalParList]
191 ConstraintIdentifier ::= Identifier
128 ASP_Identifier ::= Identifier
193 DerivationPath ::= {ConstraintIdentifier Dot}+
132 ASP_ParIdOrMacro ::= ASP_ParId&FullId | MacroSymbol
133 ASP_ParId&FullId ::= ASP_ParIdentifier [FullIdentifier]
134 ASP_ParIdentifier ::= Identifier
150 MacroSymbol ::= "<-"
197 ConstraintValue&Attributes ::= ConstraintValue ValueAttributes
198 ConstraintValue::= ConstraintExpression | MatchingSymbol | ConsRef
199 ConstraintExpression ::= Expression
200 MatchingSymbol ::= Complement | Omit | AnyValue | AnyOrOmit | ValueList | ValueRange | SuperSet | SubSet | Permutation
277 ConsRef ::= ConstraintIdentifier [ActualCrefParList]
278 ActualCrefParList ::= "(" ActualCrefPar {Comma ActualCrefPar} ")"
279 ActualCrefPar ::= Value | DataObjectIdentifier | ConsRef
213 ValueAttributes ::= [ValueLength] [**IF_PRESENT**]
214 ValueLength ::= SingleValueLength | RangeValueLength
215 SingleValueLength ::= "[" ValueBound "]"
216 ValueBound ::= Number | TS_ParIdentifier | TS_ConstIdentifier | FormalParIdentifier
217 RangeValueLength ::= "[" LowerValueBound To UpperValueBound "]"
218 LowerValueBound ::= ValueBound
37 To ::= **TO** | ".."
219 UpperValueBound ::= ValueBound | **INFINITY**

12.4 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 12.6);

d) 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.

This information shall be provided in the format shown in the following proforma:

PDU Constraint Declaration		
Constraint Name : *ConsId&ParList* **PDU Type** : *PDU_Identifier* **Derivation Path** : *[DerivationPath]* **Comments** : *[FreeText]*		
Field Name	**Field Value**	**Comments**
. . *PDU_FieldIdOrMacro* . .	. . *ConstraintValue&Attributes* . .	. . *[FreeText]* . .
Detailed Comments: *[FreeText]*		

Proforma 26 - PDU Constraint Declaration

SYNTAX DEFINITION:

190 ConsId&ParList ::= ConstraintIdentifier [FormalParList]
191 ConstraintIdentifier ::= Identifier
145 PDU_Identifier ::= Identifier
193 DerivationPath ::= {ConstraintIdentifier Dot}+
149 PDU_FieldIdOrMacro ::= PDU_FieldId&FullId | MacroSymbol
151 PDU_FieldId&FullId ::= PDU_FieldIdentifier [FullIdentifier]
152 PDU_FieldIdentifier ::= Identifier
150 MacroSymbol ::= "<-"
197 ConstraintValue&Attributes ::= ConstraintValue ValueAttributes
198 ConstraintValue::= ConstraintExpression | MatchingSymbol | ConsRef
199 ConstraintExpression ::= Expression
200 MatchingSymbol ::= Complement | Omit | AnyValue | AnyOrOmit | ValueList | ValueRange | SuperSet | SubSet | Permutation
277 ConsRef ::= ConstraintIdentifier [ActualCrefParList]
278 ActualCrefParList ::= "(" ActualCrefPar {Comma ActualCrefPar} ")"
279 ActualCrefPar ::= Value | DataObjectIdentifier | ConsRef
213 ValueAttributes ::= [ValueLength] [**IF_PRESENT**]
214 ValueLength ::= SingleValueLength | RangeValueLength
215 SingleValueLength ::= "[" ValueBound "]"
216 ValueBound ::= Number | TS_ParIdentifier | TS_ConstIdentifier | FormalParIdentifier
217 RangeValueLength ::= "[" LowerValueBound To UpperValueBound "]"
218 LowerValueBound ::= ValueBound
37 To ::= **TO** | ".."
219 UpperValueBound ::= ValueBound | **INFINITY**

EXAMPLE 43 - 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"	

12.5 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 44 - 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)		

12.6 Base constraints and modified constraints

For every PDU type definition at least one base constraint shall be specified. A base constraint specifies a set of base, or default, values for each and every field defined in the appropriate definition. There may be any number of base constraints for any particular PDU (see annex D 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 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 is not specified in the constraint, then the value in the parent constraint shall be used (*i.e.,* the value is inherited);

b) if a parameter or field and its corresponding value is specified in the constraint, then the specified value replaces the value specified in the parent constraint.

12.7 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 Specification of constraints using ASN.1

13.1 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 use of wild cards and permutations. Mechanisms to replace or omit parts of ASN.1 constraints, to be used in modified constraints, are defined.

13.2 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.

ASN.1 type constraint tables are very similar to ASP constraint tables. ASN.1 Type Constraint Declarations shall be specified in the format shown in the following proforma:

ASN.1 Type Constraint Declaration	
Constraint Name	: *ConsId&ParList*
ASN.1 Type	: *ASN1_TypeIdentifier*
Derivation Path	: *[DerivationPath]*
Comments	: *[FreeText]*
Constraint Value	
ConstraintValue&AttributesOrReplace	
Detailed Comments: *[FreeText]*	

Proforma 27 - ASN.1 Type Constraint Declaration

SYNTAX DEFINITION:

190 ConsId&ParList ::= ConstraintIdentifier [FormalParList]
191 ConstraintIdentifier ::= Identifier
55 ASN1_TypeIdentifier ::= Identifier
193 DerivationPath ::= {ConstraintIdentifier Dot}+
223 ConstraintValue&AttributesOrReplace ::= ConstraintValue&Attributes | Replacement {Comma Replacement}
197 ConstraintValue&Attributes ::= ConstraintValue ValueAttributes
198 ConstraintValue::= ConstraintExpression | MatchingSymbol | ConsRef
199 ConstraintExpression ::= Expression
200 MatchingSymbol ::= Complement | Omit | AnyValue | AnyOrOmit | ValueList | ValueRange | SuperSet | SubSet | Permutation
277 ConsRef ::= ConstraintIdentifier [ActualCrefParList]
278 ActualCrefParList ::= "(" ActualCrefPar {Comma ActualCrefPar} ")"
279 ActualCrefPar ::= Value | DataObjectIdentifier | ConsRef
213 ValueAttributes ::= [ValueLength] [**IF_PRESENT**]
214 ValueLength ::= SingleValueLength | RangeValueLength
215 SingleValueLength ::= "[" ValueBound "]"
216 ValueBound ::= Number | TS_ParIdentifier | TS_ConstIdentifier | FormalParIdentifier
217 RangeValueLength ::= "[" LowerValueBound To UpperValueBound "]"
218 LowerValueBound ::= ValueBound

37 To ::= **TO** | ".."
219 UpperValueBound ::= ValueBound | **INFINITY**
224 Replacement ::= (**REPLACE** ReferenceList **BY** ConstraintValue&Attributes) | (**OMIT** ReferenceList)
225 ReferenceList ::= (ArrayRef | ComponentIdentifier | ComponentPosition) {ComponentReference}

This proforma is used for ASN.1 Types in the same way that the ASN.1 PDU Constraint Declaration proforma is used (see 13.4).

13.3 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 12.6 and 13.6),

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 11.6 can be used in ASN.1 constraints.

ASN.1 ASP Constraint Declarations shall be specified in the format shown in the following proforma:

ASN.1 ASP Constraint Declaration
Constraint Name : *ConsId&ParList* **ASP Type** : *ASP_Identifier* **Derivation Path** : *[DerivationPath]* **Comments** : *[FreeText]*
Constraint Value
ConstraintValue&AttributesOrReplace
Detailed Comments : *[FreeText]*

Proforma 28 - ASN.1 ASP Constraint Declaration

SYNTAX DEFINITION:

190 ConsId&ParList ::= ConstraintIdentifier [FormalParList]
191 ConstraintIdentifier ::= Identifier
128 ASP_Identifier ::= Identifier
193 DerivationPath ::= {ConstraintIdentifier Dot}+
223 ConstraintValue&AttributesOrReplace ::= ConstraintValue&Attributes | Replacement {Comma Replacement}
197 ConstraintValue&Attributes ::= ConstraintValue ValueAttributes
198 ConstraintValue::= ConstraintExpression | MatchingSymbol | ConsRef
199 ConstraintExpression ::= Expression
200 MatchingSymbol ::= Complement | Omit | AnyValue | AnyOrOmit | ValueList | ValueRange | SuperSet | SubSet | Permutation
277 ConsRef ::= ConstraintIdentifier [ActualCrefParList]
278 ActualCrefParList ::= "(" ActualCrefPar {Comma ActualCrefPar} ")"
279 ActualCrefPar ::= Value | DataObjectIdentifier | ConsRef

213 ValueAttributes ::= [ValueLength] [**IF_PRESENT**]
214 ValueLength ::= SingleValueLength | RangeValueLength
215 SingleValueLength ::= "[" ValueBound "]"
216 ValueBound ::= Number | TS_ParIdentifier | TS_ConstIdentifier | FormalParIdentifier
217 RangeValueLength ::= "[" LowerValueBound To UpperValueBound "]"
218 LowerValueBound ::= ValueBound
37 To ::= **TO** | ".."
219 UpperValueBound ::= ValueBound | **INFINITY**
224 Replacement ::= (**REPLACE** ReferenceList **BY** ConstraintValue&Attributes) | (**OMIT** ReferenceList)
225 ReferenceList ::= (ArrayRef | ComponentIdentifier | ComponentPosition) {ComponentReference}

This proforma is used for ASN.1 ASPs in the same way that the ASN.1 PDU Constraint Declaration proforma is used (see 13.4).

13.4 ASN.1 PDU Constraint Declarations

PDU constraint tables are very similar to ASP constraint tables. ASN.1 PDU Constraint Declarations shall be specified in the format shown in the following proforma:

ASN.1 PDU Constraint Declaration	
Constraint Name : *ConsId&ParList* **PDU Type** : *PDU_Identifier* **Derivation Path** : *[DerivationPath]* **Comments** : *[FreeText]*	
Constraint Value	
ConstraintValue&AttributesOrReplace	
Detailed Comments: *[FreeText]*	

Proforma 29 - ASN.1 PDU Constraint Declaration

SYNTAX DEFINITION:

190 ConsId&ParList ::= ConstraintIdentifier [FormalParList]
191 ConstraintIdentifier ::= Identifier
145 PDU_Identifier ::= Identifier
193 DerivationPath ::= {ConstraintIdentifier Dot}+
223 ConstraintValue&AttributesOrReplace ::= ConstraintValue&Attributes | Replacement {Comma Replacement}
197 ConstraintValue&Attributes ::= ConstraintValue ValueAttributes
198 ConstraintValue::= ConstraintExpression | MatchingSymbol | ConsRef
199 ConstraintExpression ::= Expression
200 MatchingSymbol ::= Complement | Omit | AnyValue | AnyOrOmit | ValueList | ValueRange | SuperSet | SubSet | Permutation
277 ConsRef ::= ConstraintIdentifier [ActualCrefParList]
278 ActualCrefParList ::= "(" ActualCrefPar {Comma ActualCrefPar} ")"
279 ActualCrefPar ::= Value | DataObjectIdentifier | ConsRef
213 ValueAttributes ::= [ValueLength] [**IF_PRESENT**]
214 ValueLength ::= SingleValueLength | RangeValueLength
215 SingleValueLength ::= "[" ValueBound "]"
216 ValueBound ::= Number | TS_ParIdentifier | TS_ConstIdentifier | FormalParIdentifier
217 RangeValueLength ::= "[" LowerValueBound To UpperValueBound "]"
218 LowerValueBound ::= ValueBound

37 To ::= **TO** | ".."
219 UpperValueBound ::= ValueBound | **INFINITY**
224 Replacement ::= (**REPLACE** ReferenceList **BY** ConstraintValue&Attributes) | (**OMIT** ReferenceList)
225 ReferenceList ::= (ArrayRef | ComponentIdentifier | ComponentPosition) {ComponentReference}

13.5 Parameterized ASN.1 constraints

ASN.1 constraints may be parameterized (see 12.5).

13.6 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:

224 Replacement ::= (**REPLACE** ReferenceList **BY** ConstraintValue&Attributes) | (**OMIT** ReferenceList)
225 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.

13.7 Formal parameter lists in modified ASN.1 constraints

The requirements of 12.7 also apply to modified ASN.1 constraints.

13.8 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 45 - Field values in an ASN.1 PDU constraint. Assume the type definition:

ASN.1 PDU Type Definition			
PDU Name : XY_PDU **PCO Type** : **Comment** :			
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 or
-- field_4, since both are OPTIONAL.
```

14 The Dynamic Part

14.1 Introduction

The Dynamic Part contains the main body of the test suite: the Test Case, the Test Step and the Default behaviour descriptions.

14.2 Test Case dynamic behaviour

14.2.1 Specification of the Test Case Dynamic Behaviour table

14.2.1.1 The title of the table shall be "Test Case Dynamic Behaviour"

14.2.1.2 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 8.2, 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 standard (if any) or equivalent part of the test suite standard (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 14.4);

14.2.1.3 The body of the table shall display the following columns and corresponding information:

a) an (optional) line number column (see 14.2.4),

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 14.14);

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 14.6);

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 11);

e) a verdict column,

where verdict or result information is placed in association with TTCN statements in the behaviour tree (see 14.17);

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.

14.2.1.4 An (optional) footer can contain detailed comments.

14.2.2 The Test Case Dynamic Behaviour proforma

The Test Case dynamic behaviour shall be provided in the format shown in the following proforma:

Test Case Dynamic Behaviour						
Test Case Name : *TestCaseIdentifier* **Group** : *TestGroupReference* **Purpose** : *FreeText* **Default** : *[DefaultReference]* **Comments** : *[FreeText]*						
Nr	**Label**	**Behaviour Description**	Label	**Constraints Ref**	**Verdict**	**Comments**
1 2 . . . . . n	. . *[Label]*	. . *StatementLine* . . *TreeHeader* *StatementLine* . .	. . Alternative position for the Label column . .	. . *[ConstraintReference]*	. . *[Verdict]*	. . *[[FreeText]]* . . . . . .
Detailed Comments: *[FreeText]*						

Proforma 30 - Test Case Dynamic Behaviour

The alternative position of the label column is shown in dotted lines.

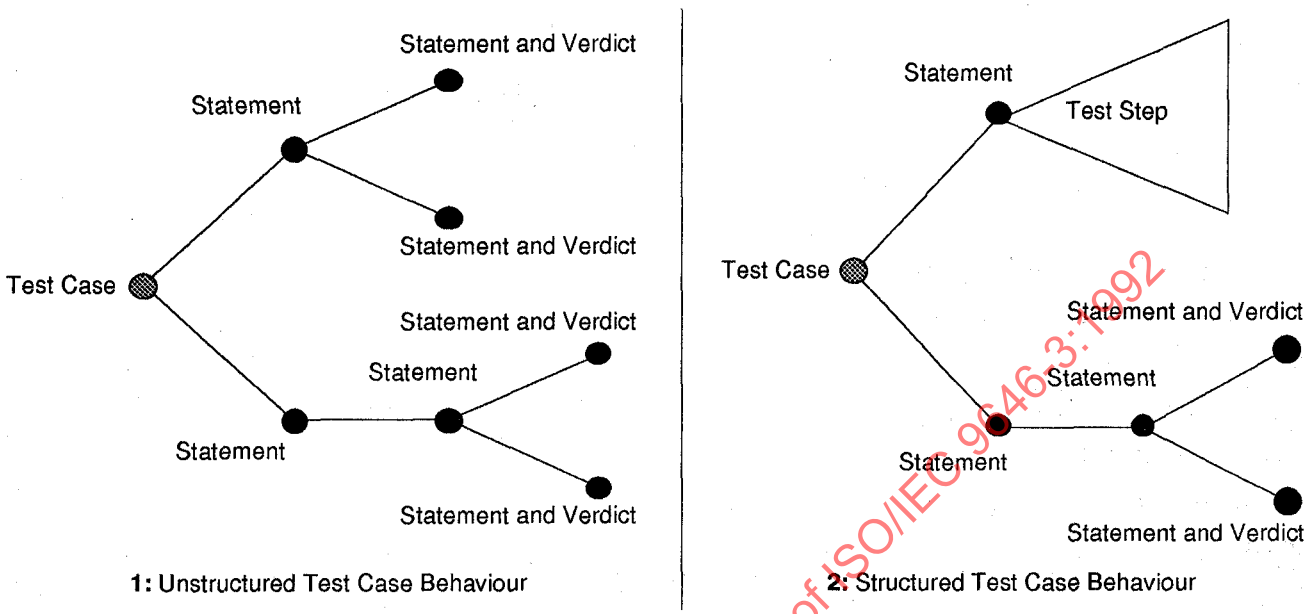
Column headers of this proforma can be abbreviated to: **L, Cref, V** and **C**. This enables the behaviour tree column to be as wide as possible in cases of physical paper size limitations.

SYNTAX DEFINITION:

```
233 TestCaseIdentifier ::= Identifier
235 TestGroupReference ::= [SuiteIdentifier "/"] {TestGroupIdentifier "/"}
238 DefaultReference ::= DefaultIdentifier [ActualParList]
299 ActualParList ::= "(" ActualPar {Comma ActualPar } ")"
300 ActualPar ::= Value | PCO_Identifier
274 Label ::= Identifier
286 StatementLine ::= (Event [Qualifier] [AssignmentList] [TimerOps]) | (Qualifier [AssignmentList] [TimerOps]) |
    (AssignmentList [TimerOps]) | TimerOps | Construct | ImplicitSend
264 TreeHeader ::= TreeIdentifier [FormalParList]
265 TreeIdentifier ::= Identifier
266 FormalParList ::= "(" FormalPar&Type {SemiColon FormalPar&Type} ")"
267 FormalPar&Type ::= FormalParIdentifier {Comma FormalParIdentifier} Colon FormalParType
268 FormalParIdentifier ::= Identifier
269 FormalParType ::= Type | PCO_TypeIdentifier | PDU
331 Type ::= PredefinedType | ReferenceType
277 ConsRef ::= ConstraintIdentifier [ActualCrefParList ]
278 ActualCrefParList ::= "(" ActualCrefPar {Comma ActualCrefPar} ")"
279 ActualCrefPar ::= Value | DataObjectIdentifier | ConsRef
281 Verdict ::= Pass | Fail | Inconclusive | Result
282 Pass ::= PASS | P | "(" PASS ")" | "(" P ")"
283 Fail ::= FAIL | F | "(" FAIL ")" | "(" F ")"
284 Inconclusive ::= INCONC | I | "(" INCONC ")" | "(" I ")"
285 Result ::= Identifier
```

14.2.3 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:



1: Unstructured Test Case Behaviour

2: Structured Test Case Behaviour

Figure 3 - Test Case Behaviour Structure

In TTCN this explicit modularization is expressed using Test Steps and the ATTACH construct.

14.2.4 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;

NOTES

1 The line numbers can be used for logging purposes, to record unambiguously which behaviour line was executed.

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 46 - Printing long behaviour lines

46.1 Recommended style:

Nr	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 alternativestatement line	Ref2		

46.2 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 alternativestatement line	Ref2		

14.3 Test Step dynamic behaviour

14.3.1 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 14.7). 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 8.3), 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.

14.3.2 The Test Step Dynamic Behaviour proforma

The Test Step dynamic behaviour shall be provided in the format shown in the following proforma:

Test Step Dynamic Behaviour

Test Step Name : *TestStepId&ParList*
Group : *TestStepGroupReference*
Objective : *FreeText*
Default : *[DefaultReference]*
Comments : *[FreeText]*

Nr	Label	Behaviour Description	Label	Constraints Ref	Verdict	Comments
1 2 . . . . n	. . *[Label]*	. . *StatementLine* . . *TreeHeader* *StatementLine* . .	. . Alternative position for the Label column . .	. . *[ConstraintReference]*	. . *[Verdict]*	. . *[FreeText]* . . . . .

Detailed Comments: *[FreeText]*

Proforma 31 - 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:

245 TestStepId&ParList ::= TestStepIdentifier [FormalParList]
246 TestStepIdentifier ::= Identifier
266 FormalParList ::= "(" FormalPar&Type {SemiColon FormalPar&Type} ")"
267 FormalPar&Type ::= FormalParIdentifier {Comma FormalParIdentifier} Colon FormalParType
268 FormalParIdentifier ::= Identifier
269 FormalParType ::= Type | PCO_TypeIdentifier | **PDU**
331 Type ::= PredefinedType | ReferenceType
248 TestStepGroupReference ::= [SuiteIdentifier "/"] {TestStepGroupIdentifier "/"}
238 DefaultReference ::= DefaultIdentifier [ActualParList]
299 ActualParList ::= "(" ActualPar {Comma ActualPar } ")"
300 ActualPar ::= Value | PCO_Identifier
274 Label ::= Identifier
286 StatementLine ::= (Event [Qualifier] [AssignmentList] [TimerOps]) | (Qualifier [AssignmentList] [TimerOps]) | (AssignmentList [TimerOps]) | TimerOps | Construct | ImplicitSend
264 TreeHeader ::= TreeIdentifier [FormalParList]
265 TreeIdentifier ::= Identifier
266 FormalParList ::= "(" FormalPar&Type {SemiColon FormalPar&Type} ")"
267 FormalPar&Type ::= FormalParIdentifier {Comma FormalParIdentifier} Colon FormalParType
268 FormalParIdentifier ::= Identifier
269 FormalParType ::= Type | PCO_TypeIdentifier | **PDU**
331 Type ::= PredefinedType | ReferenceType
277 ConsRef ::= ConstraintIdentifier [ActualCrefParList]
278 ActualCrefParList ::= "(" ActualCrefPar {Comma ActualCrefPar} ")"
279 ActualCrefPar ::= Value | DataObjectIdentifier | ConsRef
281 Verdict ::= Pass | Fail | Inconclusive | Result
282 Pass ::= **PASS** | **P** | "(" **PASS** ")" | "(" **P** ")"
283 Fail ::= **FAIL** | **F** | "(" **FAIL** ")" | "(" **F** ")"
284 Inconclusive ::= **INCONC** | **I** | "(" **INCONC** ")" | "(" **I** ")"
285 Result ::= Identifier

14.4 Default dynamic behaviour

14.4.1 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) the behaviour description shall consist of only one tree (*i.e.,* no local trees);

c) the tree in the behaviour description shall not use tree attachment (*i.e.,* Default behaviour trees shall not attach Test Steps).

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 substitution of these parameters apply as described for tree attachment (see 14.13).

14.4.2 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 (see 8.4), 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.

14.4.3 The Default Dynamic Behaviour proforma

The Default dynamic behaviour shall be provided in the format shown in the following proforma:

Default Dynamic Behaviour

Default Name : *DefaultId&ParList*
Group : *DefaultGroupReference*
Objective : *FreeText*
Comments : *[FreeText]*

Nr	Label	Behaviour Description	Label	Constraints Ref	Verdict	Comments
1 2 . . . . . . n	. . *[Label]*	. . *StatementLine*	. . Alternative position for the Label column	. . *[ConstraintReference]*	. . *[Verdict]*	. . *[FreeText]* . . . .

Detailed Comments: *[FreeText]*

Proforma 32 - 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:

256 DefaultId&ParList ::= DefaultIdentifier [FormalParList]
257 DefaultIdentifier ::= Identifier
266 FormalParList ::= "(" FormalPar&Type {SemiColon FormalPar&Type} ")"
267 FormalPar&Type ::= FormalParIdentifier {Comma FormalParIdentifier} Colon FormalParType
268 FormalParIdentifier ::= Identifier
269 FormalParType ::= Type | PCO_TypeIdentifier | **PDU**
331 Type ::= PredefinedType | ReferenceType
238 DefaultReference ::= DefaultIdentifier [ActualParList]
299 ActualParList ::= "(" ActualPar {Comma ActualPar } ")"
300 ActualPar ::= Value | PCO_Identifier
274 Label ::= Identifier

```
286 StatementLine ::= (Event [Qualifier] [AssignmentList] [TimerOps]) | (Qualifier [AssignmentList] [TimerOps]) |
    (AssignmentList [TimerOps]) | TimerOps | Construct | ImplicitSend
277 ConsRef ::= ConstraintIdentifier [ActualCrefParList ]
278 ActualCrefParList ::= "(" ActualCrefPar {Comma ActualCrefPar} ")"
279 ActualCrefPar ::= Value | DataObjectIdentifier | ConsRef
281 Verdict ::= Pass | Fail | Inconclusive | Result
282 Pass ::= PASS | P | "(" PASS ")" | "(" P ")"
283 Fail ::= FAIL | F | "(" FAIL ")" | "(" F ")"
284 Inconclusive ::= INCONC | I | "(" INCONC ")" | "(" I ")"
285 Result ::= Identifier
```

14.5 The 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.

14.6 The 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 47 - 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 48 - Alternative TTCN statements:

```
CONSTRUCT_A1
STATEMENT_A2
EVENT_A3
```

EXAMPLE 49 - 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:

271 Line ::= $Line Indentation StatementLine

EXAMPLE 50 - $Line [6] +R1_POSTAMBLE

14.7 Tree names and parameter lists

14.7.1 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:

261 RootTree ::= {BehaviourLine}+
262 LocalTree ::= Header {BehaviourLine}+

14.7.2 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 sub-list. When such a sub-list is used, the formal parameters within the sub-list shall be separated from each other by a comma. The final formal parameter in the sub-list shall be followed by a colon and the formal parameter's type.

When there is more than one formal parameter and type pair (or more than one sub-list 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 51 - A Test Step using formal parameters: EXAMPLE_TREE (L:TSAP; X:INTEGER; Y:INTEGER)

EXAMPLE 52 - A Test Step using a formal parameters with a sub-list: EXAMPLE_TREE (L:TSAP; X, Y:INTEGER)

14.8 TTCN statements

The tree notation allows the specification of test events initiated by the LT or UT (SEND and IMPLICIT SEND events), test events received by the LT or UT (RECEIVE, OTHERWISE and TIMEOUT), constructs (GOTO, ATTACH and REPEAT) 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.

14.9 TTCN test events

14.9.1 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 10.8 and 14.9.5.2.

SYNTAX DEFINITION:

289 Send ::= [PCO_Identifier | FormalParIdentifier] "!" (ASP_Identifier | PDU_Identifier)
291 Receive ::= [PCO_Identifier | FormalParIdentifier] "?" (ASP_Identifier | PDU_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 53 - !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 54 - L! CONreq or L? CONind

14.9.2 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 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;

b) the ASP or PDU matches the constraint reference on the event line;

c) 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.

14.9.3 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 clause 11, 12 and 13 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.

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.

14.9.4 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 55 - !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 56 - ?A_PDU (VAR:=A_PDU.FIELD)

where VAR may be used on event lines subsequent to receipt of A_PDU.

14.9.5 Execution of the behaviour tree

14.9.5.1 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 57 - 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:

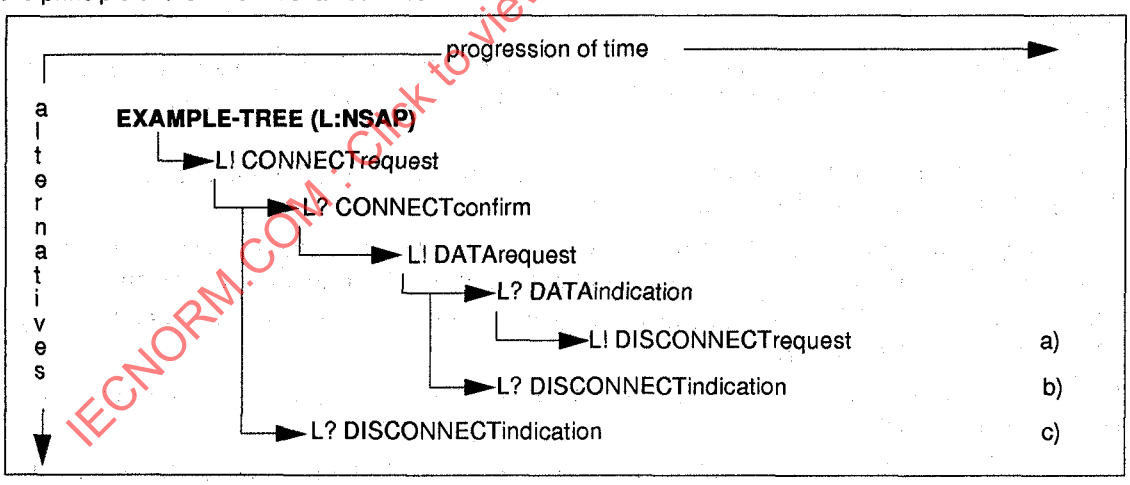
a) CONNECTrequest, CONNECTconfirm, DATArequest, DATAindication, DISCONNECTrequest;

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:



There are no lines, arrows or leaf names in TTCN. The behaviour tree of the previous example would be represented as follows:

EXAMPLE 58 - 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 :
Comments : NOTE - This example can be simplified by using Defaults

Nr	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

14.9.5.2 The 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.

14.9.5.3 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 MOT (Means of Testing) 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 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 queues

EXAMPLE 59 - An incomplete set of RECEIVE events

```
PARTIAL_TREE
!A START T
   ?B
   ?TIMEOUT T
      !C
         ?D          PASS
```

a)

```
PARTIAL_TREE
!A START T
   ?B
   ? OTHERWISE          FAIL
   ?TIMEOUT T
      !C
         ?D             PASS
         ?OTHERWISE     FAIL
```

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:

14.9.6 The 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. For this purpose, the implicit send event is defined, with the following syntax:

SYNTAX DEFINITION:

290 ImplicitSend ::= "<" **IUT** "!" (ASP_Identifier | PDU_Identifier) ">"

The **IUT** in the syntax takes the place of the PCOidentifier 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 standard(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 1 - 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 standard.

When an IMPLICIT SEND event is specified, the associated internal events within the IUT necessary to meet the requirements of the standard 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 coded on 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 60 - EXAMPLE use of IMPLICIT SEND

Test Case Dynamic Behaviour					
Test Case Name : IMP1 **Group** : TTCN_EXAMPLES/IMPLICIT_SEND1/ **Purpose** : A partial tree to illustrate the use of IMPLICIT SEND. **Default** : **Comments** :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
: 5 6 7 : 12 :		: <IUT ! CR > L? CR L! CC : L? OTHERWISE :	 CR1 CR1 CC1		

14.9.7 The 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:

292 Otherwise ::= [PCO_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.

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 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 61 - 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 62 - 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.

14.9.8 The 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:

293 Timeout ::= "?" **TIMEOUT** [TimerIdentifier]

EXAMPLE 63 - Use of TIMEOUT:

	?TIMEOUT T			

Since TIMEOUT events are not RECEIVE events they are not rendered unreachable by previously listed OTHERWISE alternatives.

14.10 TTCN expressions

14.10.1 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:

303 Expression ::= SimpleExpression [RelOp SimpleExpression]
304 SimpleExpression ::= Term {AddOp Term}
305 Term ::= Factor {MultiplyOp Factor}
306 Factor ::= [UnaryOp] Primary
307 Primary ::= Value | DataObjectReference | OpCall | SelectExprIdentifier | "(" Expression ")"
336 Value ::= LiteralValue | ASN1_Value
337 LiteralValue ::= Number | BooleanValue | Bstring | Hstring | Ostring | Cstring
338 Number ::= (NonZeroNum {Num}) | **0**
339 NonZeroNum ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
340 Num ::= **0** | NonZeroNum
341 BooleanValue ::= **TRUE** | **FALSE**
342 Bstring ::= "'" {Bin | Wildcard} "'" **B**
343 Bin ::= **0** | **1**
344 Hstring ::= "'" {Hex | Wildcard} "'" **H**
345 Hex ::= Num | **A** | **B** | **C** | **D** | **E** | **F**
346 Ostring ::= "'" {Oct | Wildcard} "'" **O**
347 Oct ::= Hex Hex
348 Cstring ::= """ {Char | Wildcard | "\"} """
349 Char ::= /* REFERENCE - A character defined by the relevant character string type */
350 Wildcard ::= AnyOne | AnyOrNone
351 AnyOne ::= "?"
352 AnyOrNone ::= "*"
308 DataObjectReference ::= DataObjectIdentifier {ComponentReference}
309 DataObjectIdentifier ::= TS_ParIdentifier | TS_ConstIdentifier | TS_VarIdentifier | TC_VarIdentifier | FormalParIdentifier | ASP_Identifier | PDU_Identifier
310 ComponentReference ::= RecordRef | ArrayRef | BitRef
311 RecordRef ::= Dot (ComponentIdentifier | PDU_Identifier | StructIdentifier | ComponentPosition)
312 ComponentIdentifier ::= ASP_ParIdentifier | PDU_FieldIdentifier | ElemIdentifier | ASN1_Identifier
314 ComponentPosition ::= "("Number")"
315 ArrayRef ::= Dot "[" ComponentNumber "]"
316 ComponentNumber ::= Expression
317 BitRef ::= Dot (BitIdentifier | "[" BitNumber "]")
318 BitIdentifier ::= Identifier
319 BitNumber ::= Expression
320 OpCall ::= TS_OpIdentifier (ActualParList | "(" ")")
321 AddOp ::= "+" | "-" | **OR**
322 MultiplyOp ::= "*" | "/" | **MOD** | **AND**
323 UnaryOp ::= "+" | "-" | **NOT**
324 RelOp::= "=" | "<" | ">" | "<>" | ">=" | "<="

14.10.2 References for ASN.1 defined data objects

In order to refer to components of structured data objects the following access mechanisms are provided:

a) a reference to a component of one of the following types: SEQUENCE, SET and CHOICE is constructed using a dot notation; *i.e.,* appending a dot and the name (component identifier) of the desired component to the data object identifier; the component identifier shall be used if specified;

b) references to unnamed components are constructed by giving the position of the component within the type definition in parentheses; numbering of such components of such a type starts with zero.

SYNTAX DEFINITION:

```
310 ComponentReference ::= RecordRef | ArrayRef | BitRef
311 RecordRef ::= Dot (ComponentIdentifier | PDU_Identifier | StructIdentifier | ComponentPosition)
312 ComponentIdentifier ::= ASP_ParIdentifier | PDU_FieldIdentifier | ElemIdentifier | ASN1_Identifier
314 ComponentPosition ::= "("Number")"
315 ArrayRef ::= Dot "[" ComponentNumber "]"
316 ComponentNumber ::= Expression
```

An index enclosed in square brackets is used to refer to a component of an ASN.1 SEQUENCE OF or SET OF type.

The first component has the number zero.

The expression shall evaluate to a non-negative INTEGER.

EXAMPLE 64 - Component references

```
Example_type ::= SEQUENCE {
                                field_1    INTEGER,
                                field_2    BOOLEAN,
                                           OCTET STRING  }
```

If var1 is of ASN.1 type Example_type, then we could write:

var1.field_1	which refers to the first INTEGER field
var1.(3)	which refers to the third (unnamed) field

EXAMPLE 65 - PDU Field references

```
XY_PDUtype ::= SEQUENCE  {
                                :
                                user_data     OCTET STRING,
                                :    }
```

On a statement line that contains XY_PDUtype, we could write: L? XY_PDU (buffer := XY_PDUtype.user_data)

The index notation is used to refer to elements (bits) of the ASN.1 BITSTRING type. BITSTRING is assumed to be defined as SEQUENCE OF {BOOLEAN}. If certain bits of a BITSTRING are associated with an identifier (named bit) then either the dot notation or this identifier shall be used to refer to the bit.

SYNTAX DEFINITION:

```
317 BitRef ::= Dot (BitIdentifier | "[" BitNumber "]")
318 BitIdentifier ::= Identifier
319 BitNumber ::= Expression
```

The leftmost bit has the number zero.

The expression shall evaluate to a non-negative INTEGER.

EXAMPLE 66 - Component references on BITSTRING types:

```
B_type ::= BIT STRING { ack(0), poll(3) }
```

where bit zero is called "ack" and bit three is called "poll".

If b_str is of ASN.1 type B_type, then we could write:

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.

ISO/IEC 8824 defines SET and 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 and SET OF type in the same way as objects of SEQUENCE and SEQUENCE OF type, *i.e.,* referring to the components with number *i* always means a reference to the *ith* field as declared in the type. ASN.1 component identifiers comprise the name of the ASN.1 ASP parameters, ASN.1 PDU fields or sub-components of ASN.1 ASP parameters or ASN.1 PDU fields.

After an ASP or PDU 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 or SET OF by any operation in TTCN.

14.10.3 References for data objects defined using tables

The same syntax as defined in 14.10.2 shall be used to construct references to ASP parameters, PDU fields or elements of Structured Types defined in tabular form. To specify a reference to a parameter, field or element the data object identifier shall be followed by a dot (.) and a parameter, field or element identifier.

Where a parameter, field or element is defined to be a true substructure of a type defined in a Structured Type table, a reference to the elements in the substructure shall consist of the reference to the parameter, field or element identifier followed by a dot and the identifier of the item within that substructure.

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 structure referring to it.

If a parameter, field or element is defined to be of metatype PDU no reference shall be made to fields of that substructure.

14.10.4 Assignments

14.10.4.1 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:

301 AssignmentList ::= "(" Assignment {Comma Assignment} ")"

302 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 67 - 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.

14.10.4.2 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.*, possibly infinite 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.

14.10.5 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:

288 Qualifier ::= "[" Expression "]"

14.10.6 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 68 - 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.

EXAMPLE 69 - Using OTHERWISE, qualifiers and assignments:

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. For example:

```
PARTIAL_TREE (PCO1:XSAP; PCO2:YSAP)
PCO1? A                                                  PASS
PCO2? B [X=2]                                            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.

14.11 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.

14.12 Timer management

14.12.1 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 READ-TIMER. 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:

325 TimerOps ::= TimerOp {Comma TimerOp}
326 TimerOp ::= StartTimer | CancelTimer | ReadTimer

14.12.2 The START operation

The START operation is used to indicate that a timer should start running.

SYNTAX DEFINITION:

327 StartTimer ::= **START** TimerIdentifier ["(" TimerValue ")"]
117 *TimerIdentifier* ::= *Identifier*
329 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 70 - Uses of START timer:

the T_i are timer identifiers and the V_i 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.

14.12.3 The CANCEL operation

The CANCEL operation is used to stop a running timer.

SYNTAX DEFINITION:

328 CancelTimer ::= **CANCEL** [TimerIdentifier]
117 TimerIdentifier ::= Identifier
329 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.

Cancelling an inactive timer is a valid operation, although it does not have any effect.

EXAMPLE 71 - Some uses of CANCEL timer:

where the T_i are timer identifiers:

CANCEL
CANCEL T0
CANCEL T1, CANCEL T2
CANCEL T1, START T3

14.12.4 The READ TIMER 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:

330 ReadTimer ::= **READTIMER** TimerIdentifier "(" DataObjectReference")"
117 TimerIdentifier ::= Identifier

EXAMPLE 72 - Using READTIMER:

```
:
START TimerName (TimerVal)
    ?EVENT_A
        +Tree_A
    ?EVENT_B
        +Tree_B
    ?EVENT_C
        READTIMER  TimerName  (CurrTime)
            +Tree_C
    ?TIMEOUT  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 73 - READTIMER used in combination with other timer operations:

READTIMER T1 (PASSED_TIME), CANCEL T1

READTIMER T1 (V1), START NEW_TIMER (V1)

14.13 The ATTACH construct

14.13.1 Introduction

Trees may be attached to other trees by using the ATTACH construct, which has the syntax:

SYNTAX DEFINITION:

296 Attach ::= "+" TreeReference [ActualParList]
298 TreeReference ::= TestStepIdentifier | TreeIdentifier
299 ActualParList ::= "(" ActualPar {Comma ActualPar } ")"
300 ActualPar ::= Value | PCO_Identifier

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.

14.13.2 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

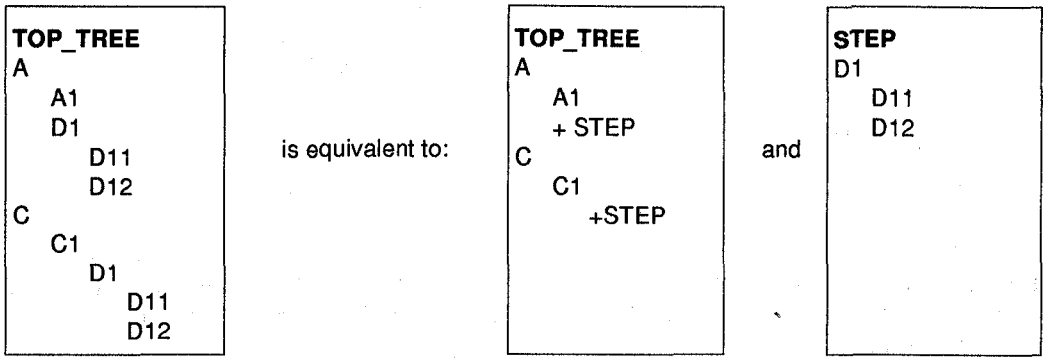
a) a Test Step Identifier denotes the attachment of a Test Step that resides in the Test Step Library; the Test Step is referenced by its unique identifier;

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.

14.13.3 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 74 - Partitioning a large tree into two smaller trees:



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.

14.13.4 The meaning of tree attachment

14.13.4.1 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:

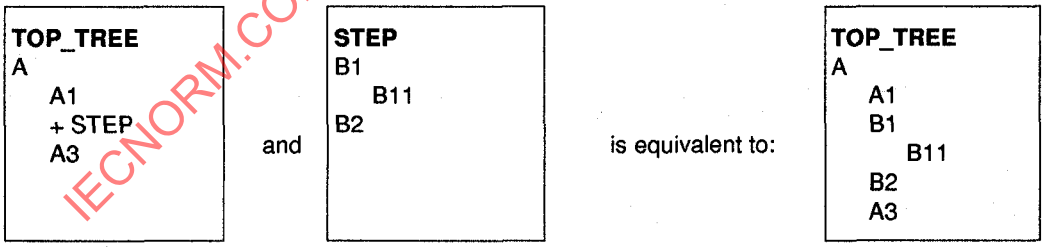
$$(A_1, \ldots, A_i, \ldots, A_n)$$

Attaching STEP in this position means expanding the TOP_TREE by inserting the Test Step STEP's top alternatives, *e.g.,* $(B_1, \ldots, B_m)$ into this sequence, yielding a new sequence:

$$(A_1, \ldots, A_{(i-1)}, B_1, \ldots, B_m, A_{(i+1)}, \ldots, A_n)$$

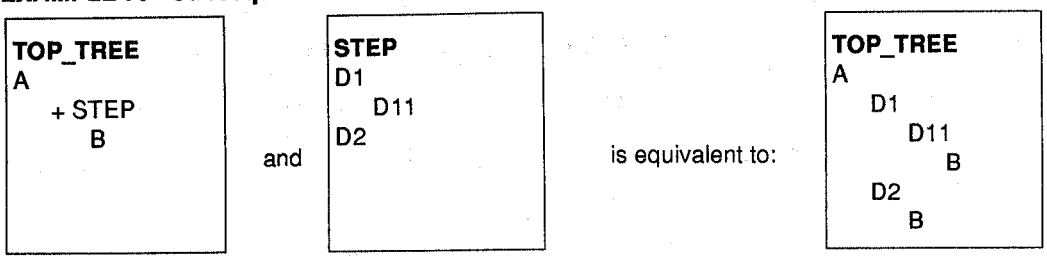
of alternatives. Any subsequent behaviour to the Bs will be attached together with them.

EXAMPLE 75 - 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 76 - 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;

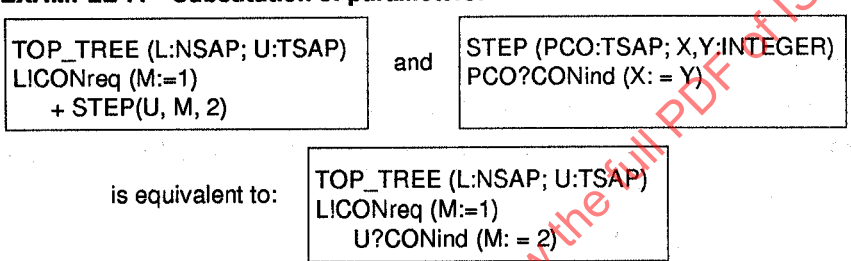
2) Actual parameters on the ATTACH of a root tree of a Test Step are substituted for all occurrences of the corresponding formals within the root tree and any local trees directly within the Test Step;

3) When a parameterized tree is attached:

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

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

EXAMPLE 77 - Substitution of parameters:



EXAMPLE 78 - Scoping rules for parameter substitution:

Test Step Dynamic Behaviour

Test Step Name : TEST_STEP_1(X,Y:INTEGER)
Group : TTCN_EXAMPLES/PARAMS/STEPS/
Objective : To illustrate scoping rules for parameter substitution.
Default :
Comments :

Nr	Label	Behaviour Description	Constraints 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.

14.13.5 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 79 - 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(PAR:A_PDU)
    :
    ! A_PDU          PAR
    :
```

14.13.6 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 80 - A legal 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
    D
        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 81 - An illegal 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
```

14.13.7 Tree attachment and Defaults

The expansion of Defaults in a tree shall be completed before this tree is attached anywhere (see 14.18.2).

NOTE - Special care has to be taken where both tree attachment and Defaults are used in a behaviour description.

14.14 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 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:

295 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 82 - Use of GOTO

Test Case Dynamic Behaviour

Test Case Name : GOTO_EX1
Group : TTCN_EXAMPLES/GOTO_EXAMPLE1/
Purpose : To illustrate use of labels and GOTO.
Default :
Comments :

Nr	Label	Behaviour Description	Constraints 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).

14.15 The 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:

297 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 14.13. 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 will normally be the last alternative of a series of TTCN statements at the same level of indentation, as allowed by 14.9.5.3 a).

NOTE 3 - The REPEAT construct is recommended, if applicable, instead of use of GOTO.

EXAMPLE 83 - Use of REPEAT (see also annex D):

Test Case Dynamic Behaviour					
Test Case Name : RPT_EX1 **Group** : TTCN_EXAMPLES/REPEAT_EXAMPLE1/ **Purpose** : To illustrate use of REPEAT. **Default** : **Comments** :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		(FLAG:=FALSE)			
2		!A	A1		
3		REPEAT STEP1 (FLAG) UNTIL [FLAG]			
4		!D	D1	PASS	
		STEP1 (F:BOOLEAN)			
5		?B (F:=TRUE)	B1		
6		?C(F:=FALSE)	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.					

14.16 The Constraints Reference

14.16.1 Purpose of the Constraints Reference column

This column allows references to be made to a specific constraint placed on an ASP or PDU. Such constraints are defined in the constraints part (see clause 11, 12 and 13). The constraints reference shall be present in conjunction with SEND, IMPLICIT SEND and RECEIVE. A constraints reference is optional if an ASP has no parameters. It shall not be present with any other kind of TTCN statement.

A constraint reference has the syntax:

SYNTAX DEFINITION:

277 ConsRef ::= ConstraintIdentifier [ActualCrefParList]
278 ActualCrefParList ::= "(" ActualCrefPar {Comma ActualCrefPar} ")"
279 ActualCrefPar ::= Value | DataObjectIdentifier | ConsRef

EXAMPLE 84 - A constraint reference without a parameter list:

	N_SAP? CR_PDU	CR1		

14.16.2 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 an element of its corresponding 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 85 - 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).

14.16.3 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.

14.17 Verdicts

14.17.1 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:

```
281  Verdict ::= Pass | Fail | Inconclusive | Result
282  Pass ::= PASS | P | "(" PASS ")" | "(" P ")"
283  Fail ::= FAIL | F | "(" FAIL ")" | "(" F ")"
284  Inconclusive ::= INCONC | I | "(" INCONC ")" | "(" I ")"
285  Result ::= Identifier
```

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.

14.17.2 Preliminary results

A predefined variable called R is available in each Test Case to store any Intermediate results. R can take the values *pass*, *fail*, *inconc* and *none*. 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 the following table:

Table 6 - 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

NOTE 2 - Thus, the order of precedence (lower → 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.

14.17.3 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 7 - 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 ISO/IEC 9646-2, 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.

14.17.4 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.

14.18 The meaning of Defaults

14.18.1 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 (+ 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 86 - Identifying a Default tree:

```
TOP_TREE
A
    A1
        A11
        C
            D
    A2
    C
        D
B
    B1
    C
        D
C
    D
```

1: the complete set of alternatives.

```
TOP_TREE
A
    A1
        A11
        + COMMON
    A2
    + COMMON
B
    B1
    + COMMON
+ COMMON

COMMON
C
    D
```

2: explicit tree attachment.

```
TOP_TREE
Default: COMMON
A
    A1
        A11
    A2
B
    B1

COMMON
C
    D
```

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, a final verdict shall be assigned to every leaf of the Default tree. If the final verdict results from an OTHERWISE statement in the Default tree, the final verdict shall be FAIL.

14.18.2 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

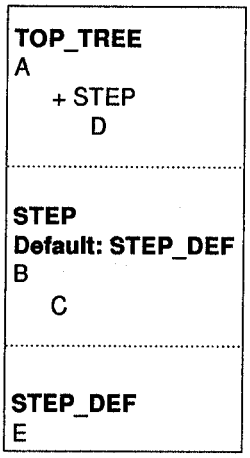
a) before the tree is expanded as an attached tree; and

b) before any of the tree's attached Test Steps are expanded.

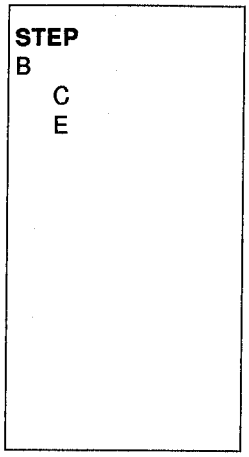
The expansion of Defaults is thus local to a single tree and comprises the attachment of the Default tree to the bottom of every set of alternatives within the tree (except the top set of alternatives for any tree other than the root tree of a Test Case).

Default expansion rules hold equally in the case where a set of alternatives contains an OTHERWISE event.

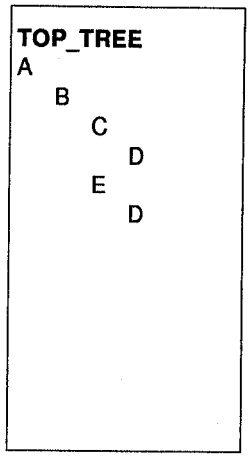
EXAMPLE 87 - Locality of a Default against a Test Step:



1: TOP_TREE attaches STEP, which has the Default STEP_DEF

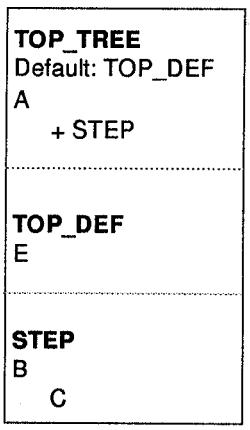


2: STEP_DEF expanded into STEP

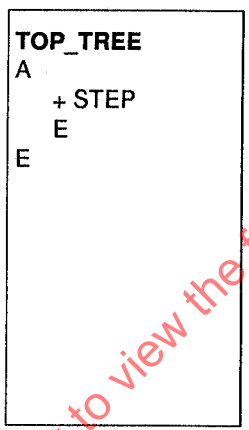


3: STEP expanded into TOP_TREE

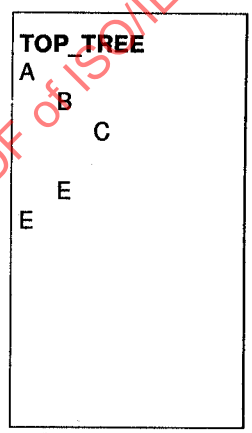
EXAMPLE 88 - Locality of a Default against a calling tree:



1: TOP_TREE attaches STEP. TOP_TREE has the Default TOP_DEF

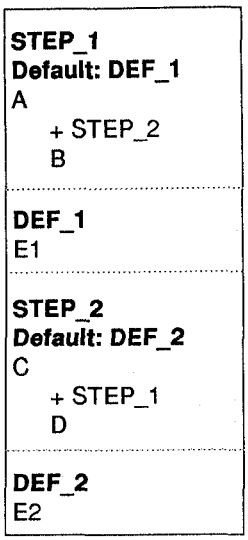


2: TOP_DEF expanded into TOP_TREE

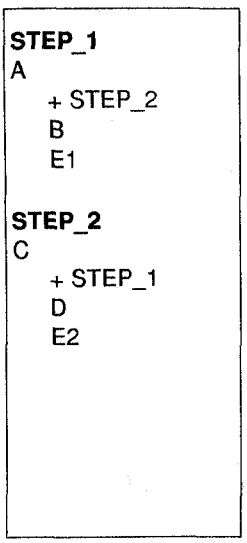


3: STEP expanded into TOP_TREE

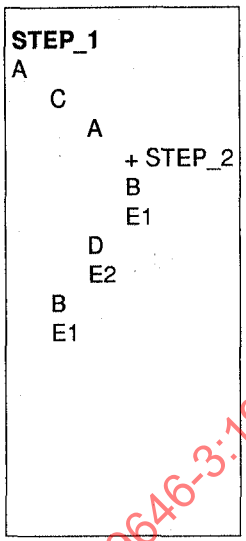
EXAMPLE 89 - A case of cyclic tree attachment:



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

14.19 Default References

Test Case and Test Step behaviours reference Default behaviour in the Default Library through the Default entry in the table header. This reference locates the Default by its unique identifier. 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; and

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.

SYNTAX DEFINITION:

238 DefaultReference ::= DefaultIdentifier [ActualParList]

The Default Identifier shall be a reference to a Default defined in the Default Library.

EXAMPLE 90 - Default reference:

90.1

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.*					
Nr	**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	DSC1	PASS	Accept

90.2

Default Dynamic Behaviour					
Default Name : DEF1(X:XSAP) **Group** : TTCN_EXAMPLES/DEFAULTS_LIB/DEFAULT_1/ **Objective** : Illustration of a simple Default. **Comments** :					
Nr	**Label**	**Behaviour Description**	**Constraints 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 Page continuation

15.1 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 91 - A continued Test Suite Parameters table:

Test Suite Parameter Declarations			
Parameter Name	**Type**	**PICS/PIXIT Ref**	**Comments**
PAR1 PAR2 PAR3	INTEGER BOOLEAN IA5String	PICS question aa PICS question bb PIXIT question cc	

Continued on next page page n

Continued from previous page page n+1

Test Suite Parameter Declarations			
Parameter Name	**Type**	**PICS/PIXIT Ref**	**Comments**
PAR4 PAR5	BOOLEAN HEXSTRING	PICS question dd PICS question ee	

15.2 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.

EXAMPLE 92 - Page splitting with re-alignment of indentation using bracketed indicators:

Test Case Dynamic Behaviour					
Test Case Name : SPLIT2 **Group** : TTCN_EXAMPLES/PAGE_SPLITTING2/ **Purpose** : To demonstrate page splitting of a dynamic behaviour table. **Default** : **Comments** :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		?A	CA		
2		?H1	CH1		
3		?J	CJ		
4		?K	CK		
		[1] [3]			

Continued on next page page n

Continued from previous page page n+1

Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
		[1] [3]			
5		?L	CL		
6		?M	CM	PASS	
7		?H2	CH2	FAIL	

As an option, test suite writers may utilize a grid at the top and bottom of the behaviour table. When a page continues onto a subsequent page the grid may be shifted to the left thus allowing the indentation to be shifted too.

EXAMPLE 93 - Re-alignment of indentation using grid indicator

Test Case Dynamic Behaviour					
Test Case Name : SPLIT2 **Group** : TTCN_EXAMPLES/PAGE_SPLITTING3/ **Purpose** : To demonstrate page splitting of a dynamic behaviour table, using a grid. **Default** : **Comments** :					
Nr	**Label**	**Behaviour Description** 0 1 2 3	**Constraints Ref**	**Verdict**	**Comments**
1		?A	CA		
2		?H1	CH1		
3		?J	CJ		
4		?K	CK		
		0 1 2 3			

Continued on next page page n

Continued from previous page page n+1

Nr	**Label**	**Behaviour Description** 1 2 3 4 ...	**Constraints Ref**	**Verdict**	**Comments**
5		?L	CL		
6		?M	CM	PASS	
7		?H2	CH2	FAIL	
		1 2 3 4 ...			

Annex A
(normative)

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 part of ISO/IEC 9646.

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 1 defines the metanotation used to specify the extended form of BNF grammar for TTCN (henceforth called BNF):

Table A.1 - The TTCN.MP Syntactic Metanotation

::=	is defined to be
|	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_Parlid ::= **$TS_Parld** 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.3 The TTCN.MP syntax productions in BNF

A.3.1 Test suite

1 Suite ::= **$Suite** SuiteId SuiteOverviewPart DeclarationsPart ConstraintsPart DynamicPart **$End_Suite**

2 SuiteId ::= **$SuiteId** SuiteIdentifier

3 SuiteIdentifier ::= Identifier

A.3.2 The Test Suite Overview

A.3.2.1 General

4 SuiteOverviewPart::= **$SuiteOverviewPart** SuiteStructure TestCaseIndex [TestStepIndex] [DefaultIndex] **$End_SuiteOverviewPart**

A.3.2.2 Test Suite Structure

5 SuiteStructure ::= **$Begin_SuiteStructure** SuiteId StandardsRef PICSref PIXITref TestMethods [Comment] Structure&Objectives [Comment] **$End_SuiteStructure**

6 StandardsRef ::= **$StandardsRef** BoundedFreeText

7 PICSref ::= **$PICSref** BoundedFreeText

8 PIXITref ::= **$PIXITref** BoundedFreeText

9 TestMethods ::= **$TestMethods** BoundedFreeText

10 Comment ::= **$Comment** [BoundedFreeText]

11 Structure&Objectives ::= **$Structure&Objectives** {Structure&Objective} **$End_Structure&Objectives**

12 Structure&Objective ::= **$Structure&Objective** TestGroupRef SelExprId Objective **$End_Structure&Objective**

13 SelExprId ::=**$SelectExprId** [SelectExprIdentifier]

A.3.2.3 Test Case Index

14 TestCaseIndex ::= **$Begin_TestCaseIndex** {CaseIndex}+ [Comment] **$End_TestCaseIndex**

15 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 */

16 Description ::= **$Description** BoundedFreeText

A.3.2.4 Test Step Index

17 TestStepIndex ::= **$Begin_TestStepIndex** {StepIndex} [Comment] **$End_TestStepIndex**

18 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 TestSteGroupReference shall be provided for each TestStep that immediately follows a TestStep Group */*/

A.3.2.5 Default Index

19 DefaultIndex ::= **$Begin_DefaultIndex** {DefIndex} [Comment] **$End_DefaultIndex**

20 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 The Declarations Part

A.3.3.1 General

21 DeclarationsPart ::= **$DeclarationsPart** Definitions Parameterization&Selection Declarations ComplexDefinitions **$End_DeclarationsPart**

A.3.3.2 Definitions

A.3.3.2.1 General

22 Definitions ::= [TS_TypeDefs] [TS_OpDefs]

A.3.3.2.2 Test Suite Type Definitions

23 TS_TypeDefs ::= **$TS_TypeDefs** [SimpleTypeDefs] [StructTypeDefs] [ASN1_TypeDefs] [ASN1_TypeRefs] **$End_TS_TypeDefs**

A.3.3.2.3 Simple Type Definitions

24 SimpleTypeDefs ::= **$Begin_SimpleTypeDefs** {SimpleTypeDef}+ [Comment] **$End_SimpleTypeDefs**

25 SimpleTypeDef ::= **$SimpleTypeDef** SimpleTypeId SimpleTypeDefinition [Comment] **$End_SimpleTypeDef**

26 SimpleTypeId ::= **$SimpleTypeId** SimpleTypeIdentifier

27 SimpleTypeIdentifier ::= Identifier

28 SimpleTypeDefinition ::= **$SimpleTypeDefinition** Type&Restriction

29 Type&Restriction ::= Type [Restriction]

30 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 */

31 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 */

32 SingleTypeLength ::= "["Number "]"

33 RangeTypeLength ::= "[" LowerTypeBound To UpperTypeBound "]"

34 IntegerRange ::= "(" LowerTypeBound To UpperTypeBound ")"

/* STATIC SEMANTICS - LowerTypeBound shall be less than UpperTypeBound */

35 LowerTypeBound ::= [Minus] Number | Minus **INFINITY**

36 UpperTypeBound ::= [Minus] Number | **INFINITY**

37 To ::= **TO** | ".."

38 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.3.3.2.4 Structured Type Definitions

39 StructTypeDefs ::= **$StructTypeDefs** {StructTypeDef}+ **$End_StructTypeDefs**

40 StructTypeDef ::= **$Begin_StructTypeDef** StructId [Comment] ElemDcls [Comment] **$End_StructTypeDef**

41 StructId ::= **$StructId** StructId&FullId

42 StructId&FullId ::= StructIdentifier [FullIdentifier]

43 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 */

44 StructIdentifier ::= Identifier

45 ElemDcls ::= **$ElemDcls** {ElemDcl}+ **$End_ElemDcls**

46 ElemDcl ::= **$ElemDcl** ElemId ElemType [Comment] **$End_ElemDcl**

47 ElemId ::= **$ElemId** ElemId&FullId

48 ElemId&FullId ::= ElemIdentifier [FullIdentifier]

49 ElemIdentifier ::= Identifier

50 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.3.3.2.5 ASN.1 Type Definitions

51 ASN1_TypeDefs ::= **$ASN1_TypeDefs** {ASN1_TypeDef}+ **$End_ASN1_TypeDefs**

52 ASN1_TypeDef ::= **$Begin_ASN1_TypeDef** ASN1_TypeId [Comment] ASN1_TypeDefinition [Comment] **$End_ASN1_TypeDef**

53 ASN1_TypeId ::= **$ASN1_TypeId** ASN1_TypeId&FullId

54 ASN1_TypeId&FullId ::= ASN1_TypeIdentifier [FullIdentifier]

55 ASN1_TypeIdentifier ::= Identifier

56 ASN1_TypeDefinition ::= **$ASN1_TypeDefinition** ASN1_Type&LocalTypes **$End_ASN1_TypeDefinition**

57 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 */

58 ASN1_Type ::= Type

/* REFERENCE - Where Type is a non-terminal defined in ISO/IEC 8824 */

/* 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 ISO/IEC 8824 */

59 ASN1_LocalType ::= Typeassignment

/* REFERENCE - Where Typeassignment is a non-terminal defined in ISO/IEC 8824 */

/* STATIC SEMANTICS - ASN.1 type definitions used within TTCN shall not use external type references as defined in ISO/IEC 8824 */

A.3.3.2.6 ASN.1 Type Definitions by Reference

60 ASN1_TypeRefs ::= **$Begin_ASN1_TypeRefs** {ASN1_TypeRef}+ [Comment] **$End_ASN1_TypeRefs**

61 ASN1_TypeRef ::= **$ASN1_TypeRef** ASN1_TypeId ASN1_TypeReference ASN1_ModuleId [Comment] **$End_ASN1_TypeRef**

/* STATIC SEMANTICS - ASN1_TypeId shall not be specified with a FullIdentifier */

62 ASN1_TypeReference ::= **$ASN1_TypeReference** TypeReference

63 TypeReference ::= typereference

/* REFERENCE - Where typereference is defined in ISO/IEC 8824:1990, subclause 8.2 */

64 ASN1_ModuleId ::= **$ASN1_ModuleId** ModuleIdentifier

65 ModuleIdentifier ::= ModuleIdentifier

/* REFERENCE - Where ModuleIdentifier is a non-terminal defined in ISO/IEC 8824 */

/* STATIC SEMANTICS - ModuleIdentifier shall be unique within the domain of interest */

A.3.3.2.7 Test Suite Operation Definitions

66 TS_OpDefs ::= **$TS_OpDefs** {TS_OpDef}+ **$End_TS_OpDefs**

67 TS_OpDef ::= **$Begin_TS_OpDef** TS_OpId TS_OpResult [Comment] TS_OpDescription [Comment] **$End_TS_OpDef**

68 TS_OpId ::= **$TS_OpId** TS_OpId&ParList

69 TS_OpId&ParList ::= TS_OpIdentifier [FormalParList]

/* STATIC SEMANTICS - A Test Suite Operation formal parameter Type shall be a PredefinedType, TS_TypeIdentifier or PDU_Identifier */

70 TS_OpIdentifier ::= Identifier

71 TS_OpResult ::= **$TS_OpResult** Type

/* STATIC SEMANTICS - Type shall be a PredefinedType, TS_TypeIdentifier or PDU_Identifier */

72 TS_OpDescription ::= **$TS_OpDescription** BoundedFreeText

A.3.3.3 Parameterization and Selection

A.3.3.3.1 General

73 Parameterization&Selection ::= [TS_ParDcls] [SelectExprDefs]

A.3.3.3.2 Test Suite Parameter Declarations

74 TS_ParDcls ::= **$Begin_TS_ParDcls** {TS_ParDcl}+ [Comment] **$End_TS_ParDcls**

75 TS_ParDcl ::= **$TS_ParDcl** TS_ParId TS_ParType PICS_PIXITref [Comment] **$End_TS_ParDcl**

76 TS_ParId ::= **$TS_ParId** TS_ParIdentifier

77 TS_ParIdentifier ::= Identifier

78 TS_ParType ::= **$TS_ParType** Type

/* STATIC SEMANTICS - Type shall be a PredefinedType, TS_TypeIdentifier or a PDU_Identifier */

79 PICS_PIXITref ::= **$PICS_PIXITref** BoundedFreeText

A.3.3.3.3 Test Case Selection Expression Definitions

80 SelectExprDefs ::= **$Begin_SelectExprDefs** {SelectExprDef}+ [Comment] **$End_SelectExprDefs**

81 SelectExprDef::= **$SelectExprDef** SelectExprId SelectExpr [Comment] **$End_SelectExprDef**

82 SelectExprId ::= **$SelectExprId** SelectExprIdentifier

83 SelectExprIdentifier ::= Identifier

84 SelectExpr ::= **$SelectExpr** SelectionExpression

85 SelectionExpression ::= Expression

/* STATIC SEMANTICS - SelectionExpression shall only contain LiteralValues, TS_ParIdentifiers, TS_ConstIdentifiers and SelectExprIdentifiers*/

/* STATIC 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.3.3.4 Declarations

A.3.3.4.1 General

86 Declarations ::= [TS_ConstDcls] [TS_VarDcls] [TC_VarDcls] PCO_Dcls [TimerDcls]

A.3.3.4.2 Test Suite Constant Declarations

87 TS_ConstDcls ::= **$Begin_TS_ConstDcls** {TS_ConstDcl}+ [Comment] **$End_TS_ConstDcls**

88 TS_ConstDcl ::= **$TS_ConstDcl** TS_ConstId TS_ConstType TS_ConstValue [Comment] **$End_TS_ConstDcl**

89 TS_ConstId ::= **$TS_ConstId** TS_ConstIdentifier

90 TS_ConstIdentifier ::= Identifier

91 TS_ConstType ::= **$TS_ConstType** Type

/* STATIC SEMANTICS - Type shall be a PredefinedType, TS_TypeIdentifier or a PDU_Identifier */

92 TS_ConstValue ::= **$TS_ConstValue** DeclarationValue

93 DeclarationValue ::= Expression

/* STATIC SEMANTICS - DeclarationValue shall only contain LiteralValues, TS_ParIdentifiers and TS_ConstIdentifiers */

/* STATIC SEMANTICS - DeclarationValue shall evaluate to an element of its declared type */

A.3.3.4.3 Test Suite Variable Declarations

94 TS_VarDcls ::= **$Begin_TS_VarDcls** {TS_VarDcl}+ [Comment] **$End_TS_VarDcls**

95 TS_VarDcl ::= **$TS_VarDcl** TS_VarId TS_VarType TS_VarValue [Comment] **$End_TS_VarDcl**

96 TS_VarId ::= **$TS_VarId** TS_VarIdentifier

97 TS_VarIdentifier ::= Identifier

98 TS_VarType ::= **$TS_VarType** Type

/* STATIC SEMANTICS - Type shall be a PredefinedType, TS_TypeIdentifier or a PDU_Identifier */

99 TS_VarValue ::= **$TS_VarValue** [DeclarationValue]

A.3.3.4.4 Test Case Variable Declarations

100 TC_VarDcls ::= **$Begin_TC_VarDcls** {TC_VarDcl}+ [Comment] **$End_TC_VarDcls**

101 TC_VarDcl ::= **$TC_VarDcl** TC_VarId TC_VarType TC_VarValue [Comment] **$End_TC_VarDcl**

102 TC_VarId ::= **$TC_VarId** TC_VarIdentifier

103 TC_VarIdentifier ::= Identifier

104 TC_VarType ::= **$TC_VarType** Type
/* STATIC SEMANTICS - Type shall be a PredefinedType, TS_TypeIdentifier or a PDU_Identifier */

105 TC_VarValue ::= **$TC_VarValue** [DeclarationValue]

A.3.3.4.5 PCO Declarations

106 PCO_Dcls ::= **$Begin_PCO_Dcls** {PCO_Dcl}+ [Comment] **$End_PCO_Dcls**
/* STATIC SEMANTICS - In accordance with ISO/IEC 9646-1 the number of PCOs shall relate to the test method used */

107 PCO_Dcl ::= **$PCO_Dcl** PCO_Id PCO_TypeId P_Role [Comment] **$End_PCO_Dcl**

108 PCO_Id ::= **$PCO_Id** PCO_Identifier

109 PCO_Identifier ::= Identifier

110 PCO_TypeId ::= **$PCO_TypeId** PCO_TypeIdentifier

111 PCO_TypeIdentifier ::= Identifier

112 P_Role ::= **$PCO_Role** PCO_Role

113 PCO_Role ::= **UT** | **LT**

A.3.3.4.6 Timer Declarations

114 TimerDcls ::= **$Begin_TimerDcls** {TimerDcl}+ [Comment] **$End_TimerDcls**

115 TimerDcl ::= **$TimerDcl** TimerId Duration Unit [Comment] **$End_TimerDcl**

116 TimerId ::= **$TimerId** TimerIdentifier

117 TimerIdentifier ::= Identifier

118 Duration ::= **$Duration** [DeclarationValue]
/* STATIC SEMANTICS - DeclarationValue shall evaluate to a non-zero positive INTEGER */

119 Unit ::= **$Unit** TimeUnit

120 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.3.3.5 ASP and PDU Type Definitions

A.3.3.5.1 General

121 ComplexDefinitions ::= [ASP_TypeDefs] PDU_TypeDefs [AliasDefs]

A.3.3.5.2 ASP Type Definitions

122 ASP_TypeDefs ::= **$ASP_TypeDefs** [TTCN_ASP_TypeDefs] [ASN1_ASP_TypeDefs] [ASN1_ASP_TypeDefsByRef] **$End_ASP_TypeDefs**

A.3.3.5.3 Tabular ASP Type Definitions

123 TTCN_ASP_TypeDefs ::= **$TTCN_ASP_TypeDefs** {TTCN_ASP_TypeDef}+ **$End_TTCN_ASP_TypeDefs**

124 TTCN_ASP_TypeDef ::= **$Begin_TTCN_ASP_TypeDef** ASP_Id PCO_Type [Comment] ASP_ParDcls [Comment] **$End_TTCN_ASP_TypeDef**

125 PCO_Type ::= **$PCO_Type** [PCO_TypeIdentifier]
/* STATIC SEMANTICS - PCO_TypeIdentifier shall be one of the PCO types used in the PCO declaration proforma */
/* STATIC SEMANTICS - If only a single PCO is defined within a test suite then PCO_TypeIdentifier is optional */

126 ASP_Id ::= **$ASP_Id** ASP_Id&FullId

127 ASP_Id&FullId ::= ASP_Identifier [FullIdentifier]

128 ASP_Identifier ::= Identifier

129 ASP_ParDcls ::= **$ASP_ParDcls** {ASP_ParDcl} **$End_ASP_ParDcls**

130 ASP_ParDcl ::= **$ASP_ParDcl** ASP_ParId ASP_ParType [Comment] **$End_ASP_ParDcl**

131 ASP_ParId ::= **$ASP_ParId** ASP_ParIdOrMacro

132 ASP_ParIdOrMacro ::= ASP_ParId&FullId | MacroSymbol
/* STATIC SEMANTICS - The MacroSymbol shall be used only in combination with a reference to a Structured Type */

133 ASP_ParId&FullId ::= ASP_ParIdentifier [FullIdentifier]

134 ASP_ParIdentifier ::= Identifier

135 ASP_ParType ::= **$ASP_ParType** Type&Attributes
/* STATIC SEMANTICS - Type shall be a PredefinedType or TS_TypeIdentifier, PDU_Identifier, or **PDU** */

A.3.3.5.4 ASN.1 ASP Type Definitions

136 ASN1_ASP_TypeDefs ::= **$ASN1_ASP_TypeDefs** {ASN1_ASP_TypeDef} **$End_ASN1_ASP_TypeDefs**

137 ASN1_ASP_TypeDef ::= **$Begin_ASN1_ASP_TypeDef** ASP_Id PCO_Type [Comment] ASN1_TypeDefinition [Comment] **$End_ASN1_ASP_TypeDef**

A.3.3.5.5 ASN.1 ASP Type Definitions by Reference

138 ASN1_ASP_TypeDefsByRef ::= **$Begin_ASN1_ASP_TypeDefsByRef** {ASN1_ASP_TypeDefByRef}+ [Comment] **$End_ASN1_ASP_TypeDefsByRef**

139 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.3.3.5.6 PDU Type Definitions

140 PDU_TypeDefs ::= **$PDU_TypeDefs** [TTCN_PDU_TypeDefs] [ASN1_PDU_TypeDefs] [ASN1_PDU_TypeDefsByRef] **$End_PDU_TypeDefs**

A.3.3.5.7 Tabular PDU Type Definitions

141 TTCN_PDU_TypeDefs ::= **$TTCN_PDU_TypeDefs** {TTCN_PDU_TypeDef}+ **$End_TTCN_PDU_TypeDefs**

142 TTCN_PDU_TypeDef ::= **$Begin_TTCN_PDU_TypeDef** PDU_Id PCO_Type [Comment] PDU_FieldDcls [Comment] **$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 */

143 PDU_Id ::= **$PDU_Id** PDU_Id&FullId

144 PDU_Id&FullId ::= PDU_Identifier [FullIdentifier]

145 PDU_Identifier ::= Identifier

146 PDU_FieldDcls ::= **$PDU_FieldDcls** {PDU_FieldDcl}+ **$End_PDU_FieldDcls**

147 PDU_FieldDcl ::= **$PDU_FieldDcl** PDU_FieldId PDU_FieldType [Comment] **$End_PDU_FieldDcl**

148 PDU_FieldId ::= **$PDU_FieldId** PDU_FieldIdOrMacro

149 PDU_FieldIdOrMacro ::= PDU_FieldId&FullId | MacroSymbol
/* STATIC SEMANTICS - The MacroSymbol shall be used only in combination with a reference to a Structured Type */

150 MacroSymbol ::= "<-"

151 PDU_FieldId&FullId ::= PDU_FieldIdentifier [FullIdentifier]

152 PDU_FieldIdentifier ::= Identifier

153 PDU_FieldType ::= **$PDU_FieldType** Type&Attributes
/* STATIC SEMANTICS - Type shall be a PredefinedType or TS_TypeIdentifier, PDU_Identifier, or **PDU** */

154 Type&Attributes::= (Type [LengthAttribute]) | **PDU**
/* STATIC 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 */

155 LengthAttribute ::= SingleLength | RangeLength

156 SingleLength ::= "[" Bound "]"

157 Bound ::= Number | TS_ParIdentifier | TS_ConstIdentifier
/* STATIC SEMANTICS - Bound shall evaluate to a non-negative INTEGER value or INFINITY */

158 RangeLength ::= "[" LowerBound To UpperBound "]"
/* STATIC SEMANTICS - LowerBound shall be less than UpperBound */

159 LowerBound ::= Bound

160 UpperBound ::= Bound | **INFINITY**

A.3.3.5.8 ASN.1 PDU Type Definitions

161 ASN1_PDU_TypeDefs ::= **$ASN1_PDU_TypeDefs** {ASN1_PDU_TypeDef} **$End_ASN1_PDU_TypeDefs**

162 ASN1_PDU_TypeDef ::= **$Begin_ASN1_PDU_TypeDef** PDU_Id PCO_Type [Comment] ASN1_TypeDefinition [Comment] **$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 */

A.3.3.5.9 ASN.1 PDU Type Definitions by Reference

163 ASN1_PDU_TypeDefsByRef ::= **$Begin_ASN1_PDU_TypeDefsByRef** {ASN1_PDU_TypeDefByRef}+ [Comment] **$End_ASN1_PDU_TypeDefsByRef**

164 ASN1_PDU_TypeDefByRef ::= **$ASN1_PDU_TypeDefByRef** PDU_Id PCO_Type ASN1_TypeReference ASN1_ModuleId [Comment] **$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.3.3.5.10 Alias Definitions

165 AliasDefs ::= **$Begin_AliasDefs** {AliasDef}+ [Comment] **$End_AliasDefs**

166 AliasDef ::= **$AliasDef** AliasId ExpandedId [Comment] **$End_AliasDef**

167 AliasId ::= **$AliasId** AliasIdentifier

168 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 */

169 ExpandedId ::= **$ExpandedId** Expansion

170 Expansion ::= ASP_Identifier | PDU_Identifier

A.3.4 The Constraints Part

A.3.4.1 General

171 ConstraintsPart ::= **$ConstraintsPart** [TS_TypeConstraints] [ASP_Constraints] [PDU_Constraints] **$End_ConstraintsPart**

A.3.4.2 Test Suite Type Constraint Declarations

172 TS_TypeConstraints ::= **$TS_TypeConstraints** [StructTypeConstraints] [ASN1_TypeConstraints] **$End_TS_TypeConstraints**

A.3.4.3 Structured Type Constraint Declarations

173 StructTypeConstraints ::= **$StructTypeConstraints** {StructTypeConstraint}+ **$End_StructTypeConstraints**

174 StructTypeConstraint ::= **$Begin_StructTypeConstraint** ConsId StructId DerivPath [Comment] ElemValues [Comment] **$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 */

175 ElemValues ::= **$ElemValues** {ElemValue}+ **$End_ElemValues**

176 ElemValue ::= **$ElemValue** ElemId ConsValue [Comment] **$End_ElemValue**

/* STATIC SEMANTICS - Parameterized Element values in a base constraint shall not be modified or explicitly omitted in a modified constraint */

A.3.4.4 ASN.1 Type Constraint Declarations

177 ASN1_TypeConstraints ::= **$ASN1_TypeConstraints** {ASN1_TypeConstraint}+ **$End_ASN1_TypeConstraints**

178 ASN1_TypeConstraint ::= **$Begin_ASN1_TypeConstraint** ConsId ASN1_TypeId DerivPath [Comment] ASN1_ConsValue [Comment] **$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 */

A.3.4.5 ASP Constraint Declarations

179 ASP_Constraints ::= **$ASP_Constraints** [TTCN_ASP_Constraints] [ASN1_ASP_Constraints] **$End_ASP_Constraints**

A.3.4.6 Tabular ASP Constraint Declarations

180 TTCN_ASP_Constraints ::= **$TTCN_ASP_Constraints** {TTCN_ASP_Constraint}+ **$End_TTCN_ASP_Constraints**

181 TTCN_ASP_Constraint ::= **$Begin_TTCN_ASP_Constraint** ConsId ASP_Id DerivPath [Comment] ASP_ParValues [Comment] **$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 - For every ASP type definition at least one base constraint shall be specified */

/* 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 */

182 ASP_ParValues ::= **$ASP_ParValues** {ASP_ParValue}+ **$End_ASP_ParValues**

183 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 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.3.4.7 ASN.1 ASP Constraint Declarations

184 ASN1_ASP_Constraints ::= **$ASN1_ASP_Constraints** {ASN1_ASP_Constraint}+ **$End_ASN1_ASP_Constraints**

185 ASN1_ASP_Constraint ::= **$Begin_ASN1_ASP_Constraint** ConsId ASP_Id DerivPath [Comment] ASN1_ConsValue [Comment] **$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 */

A.3.4.8 PDU Constraint Declarations

186 PDU_Constraints ::= **$PDU_Constraints** [TTCN_PDU_Constraints] [ASN1_PDU_Constraints] **$End_PDU_Constraints**

A.3.4.9 Tabular PDU Constraint Declarations

187 TTCN_PDU_Constraints ::= **$TTCN_PDU_Constraints** {TTCN_PDU_Constraint}+ **$End_TTCN_PDU_Constraints**

188 TTCN_PDU_Constraint ::= **$Begin_TTCN_PDU_Constraint** ConsId PDU_Id DerivPath [Comment] PDU_FieldValues [Comment] **$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 - For every PDU type definition at least one base constraint shall be specified */

/* 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 */

189 ConsId ::= **$ConsId** ConsId&ParList

190 ConsId&ParList ::= ConstraintIdentifier [FormalParList]

191 ConstraintIdentifier ::= Identifier

192 DerivPath ::= **$DerivPath** [DerivationPath]

193 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 - Constraints shall be listed 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 */

194 PDU_FieldValues ::= **$PDU_FieldValues** {PDU_FieldValue}+ **$End_PDU_FieldValues**

195 PDU_FieldValue ::= **$PDU_FieldValue** PDU_FieldId ConsValue [Comment] **$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 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 */

196 ConsValue ::= **$ConsValue** ConstraintValue&Attributes

/* STATIC SEMANTICS - ConsValue shall evaluate to an element of the type specified for the ASP parameter, PDU field or structure element */

197 ConstraintValue&Attributes ::= ConstraintValue ValueAttributes

/* 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 */

/* STATIC 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 */

198 ConstraintValue::= ConstraintExpression | MatchingSymbol | ConsRef

/* STATIC SEMANTICS - LiteralValue, TS_ParIdentifier, TS_ConstIdentifier, TS_VarIdentifier, TC_VarIdentifier, (a different) ConsRef and FormalParIdentifier may be passed as actual parameters in ConsRef */

199 ConstraintExpression ::= Expression

/* STATIC SEMANTICS - The terms in ConstraintExpression shall only contain IValue, TS_ParIdentifier, TS_ConstIdentifier, FormalParIdentifier and OpCall */

/* STATIC SEMANTICS - ConstraintExpression shall evaluate to an element of the specified type */

200 MatchingSymbol ::= Complement | Omit | AnyValue | AnyOrOmit | ValueList | ValueRange | SuperSet | SubSet | Permutation

/* NOTE - No matching symbol is considered to be a specific value */

201 Complement ::= **COMPLEMENT** ValueList

202 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 */

203 AnyValue ::= "?"

204 AnyOrOmit ::= "*"

205 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 */

206 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 parameter's, PDU field's or structure element's declared type */

207 ValRange ::= (LowerRangeBound To UpperRangeBound)

/* STATIC SEMANTICS - LowerRangeBound shall be less than UpperRangeBound */

208 LowerRangeBound ::= ConstraintExpression | Minus **INFINITY**

/* STATIC SEMANTICS - ConstraintExpression shall evaluate to a specific INTEGER value */

209 UpperRangeBound ::= ConstraintExpression | **INFINITY**

/* STATIC SEMANTICS - ConstraintExpression shall evaluate to a specific INTEGER value */

210 SuperSet ::= **SUPERSET** "(" ConstraintValue&Attributes ")"
/* STATIC SEMANTICS - The argument to SuperSet, *i.e.*, ConstraintValue&Attributes, shall be of type SET OF */

211 SubSet ::= **SUBSET** "(" ConstraintValue&Attributes ")"
/* STATIC SEMANTICS - The argument to SubSet, *i.e.*, ConstraintValue&Attributes, shall be of type SET OF */

212 Permutation ::= **PERMUTATION** ValueList
/* 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 */

213 ValueAttributes ::= [ValueLength] [**IF_PRESENT**]
/* STATIC SEMANTICS - In ASN.1 constraints IF_PRESENT shall be used only for ASP parameters or PDU fields that are declared OPTIONAL or DEFAULT */

214 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: Specifiedvalue, 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 parameter's, PDU field's or structure element's declared type */

215 SingleValueLength ::= "[" ValueBound "]"

216 ValueBound ::= Number | TS_ParIdentifier | TS_ConstIdentifier | FormalParIdentifier
/* STATIC SEMANTICS - ValueBound shall evaluate to a specific non-negative INTEGER value */

217 RangeValueLength ::= "[" LowerValueBound To UpperValueBound "]"
/* STATIC SEMANTICS - LowerValueBound shall be less than UpperValueBound */

218 LowerValueBound ::= ValueBound

219 UpperValueBound ::= ValueBound | **INFINITY**

A.3.4.10 ASN.1 PDU Constraint Declarations

220 ASN1_PDU_Constraints ::= **$ASN1_PDU_Constraints** {ASN1_PDU_Constraint}+ **$End_ASN1_PDU_Constraints**

221 ASN1_PDU_Constraint ::= **$Begin_ASN1_PDU_Constraint** ConsId PDU_Id DerivPath [Comment] ASN1_ConsValue [Comment] **$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 */

222 ASN1_ConsValue ::= **$ASN1_ConsValue** ConstraintValue&AttributesOrReplace **$End_ASN1_ConsValue**

223 ConstraintValue&AttributesOrReplace ::= ConstraintValue&Attributes | Replacement {Comma Replacement}

224 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 */

225 ReferenceList ::= (ArrayRef | ComponentIdentifier | ComponentPosition) {ComponentReference}

A.3.5 The Dynamic Part

A.3.5.1 General

226 DynamicPart ::= **$DynamicPart** TestCases [TestStepLibrary] [DefaultsLibrary] **$End_DynamicPart**

A.3.5.2 Test Cases

227 TestCases ::= **$TestCases** ({TestGroup | TestCase}+) **$End_TestCases**

228 TestGroup ::= **$TestGroup** TestGroupId {TestGroup | TestCase}+ **$End_TestGroup**

229 TestGroupId ::= **$TestGroupId** TestGroupIdentifier

230 TestGroupIdentifier ::= Identifier

231 TestCase ::= **$Begin_TestCase** TestCaseId TestGroupRef TestPurpose DefaultsRef [Comment] BehaviourDescription [Comment] **$End_TestCase**

232 TestCaseId ::= **$TestCaseId** TestCaseIdentifier

233 TestCaseIdentifier ::= Identifier

234 TestGroupRef ::= **$TestGroupRef** TestGroupReference

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

/* STATIC SEMANTICS - There shall be no white space on either side of the "/"s */

236 TestPurpose ::= **$TestPurpose** BoundedFreeText

237 DefaultsRef::= **$DefaultsRef** [DefaultReference]

238 DefaultReference ::= DefaultIdentifier [ActualParList]

A.3.5.3 Test Step Library

239 TestStepLibrary ::= **$TestStepLibrary** ({TestStepGroup | TestStep}+) **$End_TestStepLibrary**

240 TestStepGroup ::= **$TestStepGroup** TestStepGroupId {TestStepGroup | TestStep}+ **$End_TestStepGroup**

241 TestStepGroupId ::= **$TestStepGroupId** TestStepGroupIdentifier

242 TestStepGroupIdentifier ::= Identifier

243 TestStep ::= **$Begin_TestStep** TestStepId TestStepRef Objective DefaultsRef [Comment] BehaviourDescription [Comment] **$End_TestStep**

244 TestStepId ::= **$TestStepId** TestStepId&ParList

245 TestStepId&ParList ::= TestStepIdentifier [FormalParList]

246 TestStepIdentifier ::= Identifier

247 TestStepRef ::= **$TestStepRef** TestStepGroupReference

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

/* STATIC SEMANTICS - There shall be no white space on either side of the "/"s */

249 Objective ::= **$Objective** BoundedFreeText

A.3.5.4 Default Library

250 DefaultsLibrary ::= **$DefaultsLibrary** ({DefaultGroup | Default}+) **$End_DefaultsLibrary**

251 DefaultGroup ::= **$DefaultGroup** DefaultGroupId {DefaultGroup | Default}+ **$End_DefaultGroup**

252 DefaultGroupId ::= **$DefaultGroupId** DefaultGroupIdentifier

253 Default ::= **$Begin_Default** DefaultId DefaultRef Objective [Comment] BehaviourDescription [Comment] **$End_Default**

/* STATIC SEMANTICS - BehaviourDescription shall consist of only one tree (*i.e.*, no local trees) */

/* STATIC SEMANTICS - BehaviourDescription shall not use tree attachment (*i.e.*, Default behaviour trees shall not attach Test Steps) */

/* STATIC SEMANTICS - A final verdict shall be assigned to every leaf of a Default tree. If the final verdict results from an OTHERWISE statement in the Default tree the verdict shall be FAIL */

254 DefaultRef ::= **$DefaultRef** DefaultGroupReference

255 DefaultId ::= **$DefaultId** DefaultId&ParList

256 DefaultId&ParList ::= DefaultIdentifier [FormalParList]

257 DefaultIdentifier ::= Identifier

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

/* STATIC SEMANTICS - There shall be no white space on either side of the "/"s */

259 DefaultGroupIdentifier ::= Identifier

A.3.5.5 Behaviour descriptions

260 BehaviourDescription ::= **$BehaviourDescription** RootTree {LocalTree} **$End_BehaviourDescription**

261 RootTree ::= {BehaviourLine}+

262 LocalTree ::= Header {BehaviourLine}+

263 Header ::= **$Header** TreeHeader

264 TreeHeader ::= TreeIdentifier [FormalParList]

265 TreeIdentifier ::= Identifier

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

267 FormalPar&Type ::= FormalParIdentifier {Comma FormalParIdentifier} Colon FormalParType

268 FormalParIdentifier ::= Identifier

269 FormalParType ::= Type | PCO_TypeIdentifier | **PDU**

/* STATIC SEMANTICS - If a formal parameter of a Test Step is type **PDU** then specific fields in the PDU shall not be referenced in the Test Step behaviour tree */

A.3.5.6 Behaviour lines

270 BehaviourLine ::= **$BehaviourLine** LabelId Line Cref VerdictId [Comment] **$End_BehaviourLine**

271 Line ::= **$Line** Indentation StatementLine

272 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 */

273 LabelId ::= **$LabelId** [Label]

274 Label ::= Identifier

275 Cref ::= **$Cref** [ConstraintReference]

276 ConstraintReference ::= ConsRef | FormalParIdentifier

/* STATIC SEMANTICS - ConsRef shall be present in conjunction with SEND, IMPLICIT SEND and RECEIVE. A ConstraintReference is not needed for ASPs that have no parameters. It shall not be present with any other kind of TTCN statement */

/* STATIC SEMANTICS - FormalParIdentifier shall resolve to a ConsRef */

/* STATIC SEMANTICS - LiteralValue, TS_ParIdentifier, TS_ConstIdentifier, TS_VarIdentifier, TC_VarIdentifier, ConsRef and FormalParIdentifier may be passed as actual parameters to a constraint in a ConstraintReference made from a behaviour description */

/* STATIC SEMANTICS - /* ConstraintReferences on SEND events shall not include wildcards unless these are explicitly assigned specific values on the SEND event line */

277 ConsRef ::= ConstraintIdentifier [ActualCrefParList]

278 ActualCrefParList ::= "(" ActualCrefPar {Comma ActualCrefPar} ")"

/* STATIC SEMANTICS - See static semantics on production 299 */

279 ActualCrefPar ::= Value | DataObjectIdentifier | ConsRef

280 VerdictId ::= **$VerdictId** [Verdict]

281 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 */

282 Pass ::= **PASS** | **P** | "(" **PASS** ")" | "(" **P** ")"

283 Fail ::= **FAIL** | **F** | "(" **FAIL** ")" | "(" **F** ")"

284 Inconclusive ::= **INCONC** | **I** | "(" **INCONC** ")" | "(" **I** ")"

285 Result ::= Identifier

/* STATIC SEMANTICS - Result shall only be the predefined identifier R */

/* STATIC SEMANTICS - R shall not be used on the LHS of an assignment */

A.3.5.7 TTCN statements

286 StatementLine ::= (Event [Qualifier] [AssignmentList] [TimerOps]) | (Qualifier [AssignmentList] [TimerOps]) | (AssignmentList [TimerOps]) | TimerOps | Construct | ImplicitSend

287 Event ::= Send | Receive | Otherwise | Timeout

/* 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 */

288 Qualifier ::= "[" Expression "]"

/* STATIC SEMANTICS - Qualifier shall evaluate to a specific BOOLEAN value */

289 Send ::= [PCO_Identifier | FormalParIdentifier] "!" (ASP_Identifier | PDU_Identifier)

/* STATIC SEMANTICS - PCO_Identifier or FormalParIdentifier shall be present if the test suite uses more than one PCO */

/* STATIC SEMANTICS - FormalParIdentifier shall resolve to a PCO_Identifier. This must be present if the test suite uses more than one PCO */

290 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 */

291 Receive ::= [PCO_Identifier | FormalParIdentifier] "?" (ASP_Identifier | PDU_Identifier)

/* STATIC SEMANTICS - PCO_Identifier or FormalParIdentifier shall be present if the test suite uses more than one PCO */

292 Otherwise ::= [PCO_Identifier | FormalParIdentifier] "?" **OTHERWISE**

/* STATIC SEMANTICS - PCO_Identifier or FormalParIdentifier shall be present if the test suite uses more than one PCO */

293 Timeout ::= "?" **TIMEOUT** [TimerIdentifier]

294 Construct ::= GoTo | Attach | Repeat

295 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 */

/* STATIC SEMANTICS - Each label used in a GoTo construct shall be found within the Test Step in which the GoTo is used */

/* STATIC SEMANTICS - No GoTo shall be made to the first level of alternatives of local trees, Test Steps or Defaults */

296 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 - LiteralValue, TS_ParIdentifier, TS_ConstIdentifier, TS_VarIdentifier, TC_VarIdentifier, ConstraintIdentifier may and PCOs may be passed as actual parameters to an attached tree */

297 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, ConstraintIdentifier may and PCOs may be passed as actual parameters to the tree in a REPEAT statement */

298 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 */

299 ActualParList ::= "(" ActualPar {Comma ActualPar } ")"

/* STATIC SEMANTICS - The number of the actual parameters shall be the same as the number of the formal parameters */

/* STATIC SEMANTICS - Each actual parameter shall resolve to a specific value compatible with the type of its corresponding formal parameter */

/* 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 */

300 ActualPar ::= Value | PCO_Identifier

A.3.5.8 Expressions

301 AssignmentList ::= "(" Assignment {Comma Assignment} ")"

302 Assignment ::= DataObjectReference ":=" Expression

/* STATIC SEMANTICS - 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 - An expression shall contain no unbound variables */

/* STATIC SEMANTICS - The Expression on the RHS of Assignment shall evaluate to an explicit value of the type of the LHS */

303 Expression ::= SimpleExpression [RelOp SimpleExpression]

/* STATIC SEMANTICS - If both SimpleExpressions and the RelOp exist then the SimpleExpressions shall evaluate to specific values of compatible types */

/* STATIC 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 */

304 SimpleExpression ::= Term {AddOp Term}

/* STATIC 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 */

305 Term ::= Factor {MultiplyOp Factor}

/* STATIC 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 */

306 Factor ::= [UnaryOp] Primary

/* STATIC 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 */

307 Primary ::= Value | DataObjectReference | OpCall | SelectExprIdentifier | "(" Expression ")"

/* STATIC SEMANTICS - SelectExprIdentifier shall only be used within selection expressions */

308 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 */

309 DataObjectIdentifier ::=TS_ParIdentifier |TS_ConstIdentifier |TS_VarIdentifier |TC_VarIdentifier |FormalParIdentifier | ASP_Identifier | PDU_Identifier

310 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 */

311 RecordRef ::= Dot (ComponentIdentifier | PDU_Identifier | StructIdentifier | 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 be explicitly included in a RecordRef whenever 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 */

312 ComponentIdentifier ::= ASP_ParIdentifier | PDU_FieldIdentifier | ElemIdentifier | ASN1_Identifier

313 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 */

314 ComponentPosition ::= "("Number")"

315 ArrayRef ::= Dot "[" ComponentNumber "]"

316 ComponentNumber ::= Expression

/* STATIC SEMANTICS - ComponentNumber shall evaluate to a non-negative specific INTEGER value */

317 BitRef ::= Dot (BitIdentifier | "[" BitNumber "]")

318 BitIdentifier ::= Identifier

/* NOTE - BitIdentifier identifies a particular bit within an ASN.1 BIT STRING */

319 BitNumber ::= Expression

/* STATIC SEMANTICS - BitNumber shall evaluate to a non-negative specific INTEGER value */

320 OpCall ::= TS_OpIdentifier (ActualParList | "(" ")")

/* STATIC SEMANTICS - See static semantics on production 299 */

321 AddOp ::= "+" | "-" | **OR**

/* STATIC 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 */

322 MultiplyOp ::= "*" | "/" | **MOD** | **AND**

/* STATIC 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 */

323 UnaryOp ::= "+" | "-" | **NOT**

/* STATIC 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 */

324 RelOp::= "=" | "<" | ">" | "<>" | ">=" | "<="

A.3.5.9 Timer operations

325 TimerOps ::= TimerOp {Comma TimerOp}

326 TimerOp ::= StartTimer | CancelTimer | ReadTimer

327 StartTimer ::= **START** TimerIdentifier ["(" TimerValue ")"]

328 CancelTimer ::= **CANCEL** [TimerIdentifier]

329 TimerValue ::= Expression

/* STATIC SEMANTICS - Timervalue shall evaluate to a non-zero positive INTEGER */

330 ReadTimer ::= **READTIMER** TimerIdentifier "(" DataObjectReference")"

/* STATIC SEMANTICS - The DataObjectReference 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 - The DataObjectReference shall resolve to type INTEGER */

A.3.6 Types

A.3.6.1 General

331 Type ::= PredefinedType | ReferenceType

A.3.6.2 Predefined types

332 PredefinedType ::= **INTEGER** | **BOOLEAN** | **BITSTRING** | **HEXSTRING** | **OCTETSTRING** | CharacterString

333 CharacterString ::= **NumericString** | **PrintableString** | **TeletexString** | **VideotexString** | **VisibleString** | **IA5String** | **GraphicString** | **GeneralString**

A.3.6.3 Referenced types

334 ReferenceType ::= TS_TypeIdentifier | ASP_Identifier | PDU_Identifier

/* STATIC SEMANTICS - All types, other than the predefined types, used in a test suite shall be declared in the Test Suite Type definitions or ASP type definitions or PDU type definitions and referenced by name */

335 TS_TypeIdentifier ::= SimpleTypeIdentifier | StructIdentifier | ASN1_TypeIdentifier

A.3.7 Values

336 Value ::= LiteralValue | ASN1_Value

/* REFERENCE - Where ASN1_Value is Value as defined in ISO/IEC 8824 */

/* *In ISO/IEC 8824 the production DefinedValue is defined as:* DefinedValue ::= Externalvaluereference | valuereference. *For the purposes of TTCN this production is redefined to be:* DefinedValue ::= ConstraintValue&Attributes. *Note that this means that external references are not allowed in TTCN* */

/* STATIC SEMANTICS - ASN.1 Named Values shall not be used within arithmetic expressions as operands of operations */

337 LiteralValue ::= Number | BooleanValue | Bstring | Hstring | Ostring | Cstring

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

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

340 Num ::= **0** | NonZeroNum

341 BooleanValue ::= **TRUE** | **FALSE**

342 Bstring ::= "'" {Bin | Wildcard} "'" **B**

343 Bin ::= **0** | **1**

344 Hstring ::= "" {Hex | Wildcard} "" **H**

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

346 Ostring ::= "" {Oct | Wildcard} "" **O**

347 Oct ::= Hex Hex

348 Cstring ::= """ {Char | Wildcard | "\"} """

349 Char ::= */* REFERENCE - A character defined by the relevant character string type */*

/* STATIC SEMANTICS - 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 */

350 Wildcard ::= AnyOne | AnyOrNone

351 AnyOne ::= "?"

/* STATIC SEMANTICS - AnyOne shall be used only within values of string types, SEQUENCE OF and SET OF */

352 AnyOrNone ::= "*"

/* STATIC SEMANTICS - AnyOrNone shall be used only within values of string types, SEQUENCE OF and SET OF */

353 Identifier ::= Alpha{AlphaNum | Underscore}

/* 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 */

354 Alpha ::= UpperAlpha | LowerAlpha

355 AlphaNum ::= Alpha | Num

356 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**

357 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**

358 ExtendedAlphaNum ::= */* REFERENCE - A character from any character set defined in ISO/IEC 10646 */*

359 BoundedFreeText ::= */** FreeText **/*

360 FreeText ::= {ExtendedAlphaNum}

/* STATIC SEMANTICS - Free Text shall not contain the string "*/" unless preceded by backslash ("\") */

A.3.8 Miscellaneous productions

361 Comma ::= ","

362 Dot ::= "."

363 Dash ::= "-"

364 Minus ::= "-"

365 SemiColon ::= ";"

366 Colon ::= ":"

367 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 clause 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 standards. In particular, references to ASN.1 type definition modules according to ISO/IEC 8824 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 and module identifiers made within the various ASN.1 type definitions tables shall comply to the requirements for identifiers defined in ISO/IEC 8824;

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 ISO/IEC 8824 (*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;
i) PCOs;
j) Timers;
k) ASP types;
l) PDU types;
m) Structured Types;
n) Aliases;
o) ASP constraints;
p) PDU constraints;
q) Structure constraints;
r) Test Cases;
s) Test Steps;
t) Defaults.

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 keywords are listed in table A.1 and TTCN predefined identifiers are listed in table A.2. These keywords and predefined identifiers are reserved words and shall not be used as identifiers in a TTCN test suite. All TTCN keywords, predefined identifiers (predefined types, operations, variables, values) and TTCN identifiers are case sensitive.

Table A.1 - TTCN Keywords

AND	ms	REPLACE
BY	NOT	s
CANCEL	ns	START
F	OMIT	SUPERSET
FAIL	OR	SUBSET
GOTO	OTHERWISE	TIMEOUT
I	P	TO
IF_PRESENT	PASS	UNTIL
INCONC	PDU	us
IUT	PERMUTATION	UT
LT	ps	
min	READTIMER	
MOD	REPEAT	

Table A.2 - TTCN Predefined Identifiers

BITSTRING	inconc	NumericString
BOOLEAN	INFINITY	OCTETSTRING
BIT_TO_INT	INTEGER	pass
fail	INT_TO_HEX	PrintableString
FALSE	INT_TO_BIT	R
GeneralString	IS_CHOSEN	TRUE
GraphicString	IS_PRESENT	TeletexString
HEXSTRING	LENGTH_OF	VideotexString
HEX_TO_INT	none	VisibleString
IA5String	NUMBER_OF_ELEMENTS	

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.

Table A.3 - ASN.1 Reserved Words

ABSENT	FROM	PRESENT
ANY	GeneralStringGeneralizedTime	PRIVATE
APPLICATION	GraphicString	PrintableString
BEGIN	IA5String	REAL
BIT	IDENTIFIER	SEQUENCE
BOOLEAN	IMPLICIT	SET
CHOICE	IMPORT	SIZE
COMPONENT	INCLUDES	STRING
COMPONENTS	INTEGER	T61String
DEFAULT	ISO646String	TRUE
DEFINED	MAX	TeletexString
DEFINITIONS	MIN	UNIVERSAL
END	NULL	UTCTime
ENUMERATED	NumericString	VideotexString
EXPLICIT	OBJECT	VisibleString
EXPORT	OCTET	WITH
EXTERNAL	OF	
FALSE	OPTIONAL	

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.

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 or PDU 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.

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.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.

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 to 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 is 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.

Annex B
(normative)

Operational Semantics of TTCN

B.1 Introduction

This annex describes how the semantics of TTCN are defined. The definition of the TTCN semantics follows a two-phase approach. In the first phase concrete TTCN texts are mapped to a simplified structure, that can be manipulated during the second phase. This second phase can be regarded as the description of a TTCN machine, that interprets these simplified TTCN texts.

The first phase can be decomposed into three steps

a) Syntax definition;

no semantics can be assigned to a language that has no fixed (context free) grammar. Annex A describes the syntax of the TTCN by means of BNF production rules;

b) Static semantics definition;

the static semantics describe which texts generated by the context free grammar make sense (*e.g.*, used PCOs should be declared, constraints referenced from the behaviour part shall be present in the constraints part of the test suite etc.); the static semantic requirements for TTCN are specified in annex A.

c) Definition of Tree Transformation;

this third step defines how to construct an abstract evaluation tree from a concrete TTCN text that is syntactically and static semantically correct. The transformation algorithms are described in clause B.4.

In the second phase the abstract TTCN machine is defined. It is unlikely that the machine will (can) ever be built with the described architecture. Implementation problems are not considered in the definition of the dynamic, or operational, semantics. The abstract TTCN machine is described in clause B.5. The main part of the TTCN machine is a process called EVALUATE_TEST_CASE that interprets the abstract evaluation tree. The abstract evaluation tree is a parameter of this process.

B.2 Precedence

Operational semantics for the TTCN are supplied in the following clauses in two alternative notations to provide the reader with a choice of method based on personal preference: pseudo-code and natural language. Where these two notations overlap they are meant to be identical. 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, however, the pseudo-code will take precedence over the natural language text pending correction by the standards organization.

B.3 Processing of test case errors

Within the main body of this part of ISO/IEC 9646, and within this annex, there are conditions described which result in the detection of test case errors. Whenever a test case error is detected, the occurrence shall be recorded in the conformance log.

B.4 Transformation algorithms

B.4.1 Introduction

This subclause defines how to construct an abstract evaluation tree from a TTCN Test Case, that is correct with respect to the syntax and static semantics. This is done in three subsequent steps

a) appending of Default behaviours;

b) removal of REPEAT constructs;

c) expansion of attached trees.

The transformations are described by means of algorithms in a pseudo programming language. In addition to this a natural language description is provided, to explain the working of the algorithms.

B.4.2 Appending default behaviour

The actual appending of Defaults is done by adding the construct "+ DefaultReference" to the end of each set of alternatives in the Test Case. For the following pseudo code, Level represents the alternatives in the current behaviour tree. If A_i is an alternative in a set of M alternatives in the current behaviour tree then Level is the ordered set (A_1, A_2, . . ., A_m) where each A_i is either an event, a pseudo-event or a construct. Associated with each A_i is a lower Level of zero or more alternatives. Thus each A_i represents a subtree of the original tree.

```
• function APPEND_DEFAULT (TestCaseOrStep, Tree, Level) :BOOLEAN
  begin
    if (TestCaseOrStep.DefaultReference <> empty) then
      begin
      (* Append Default to first level of first test case tree *)
        if  ((TestCaseOrStep is a TestCase)) and
            (Tree=ROOT_TREE(TestCaseOrStep) and
            (Level=FIRST_LEVEL(Tree)))  then
            APPEND('+' TestCaseOrStep.DefaultReference, Level);
      (* Append Default to all levels below first level *)
        if (Level<> FIRST_LEVEL(Tree) then
            APPEND('+' TestCaseOrStep.DefaultReference, Level)
      end;
    RETURN(TRUE);
  end
```

Appending all Defaults in a Test Case or Test Step can be done by traversing the whole behaviour tree and using APPEND_DEFAULT at every level of alternatives. This can be done with the following recursive procedure, called with: TestCaseOrStep:=TestCase, Tree:=ROOT_TREE(TestCase), and Level:=FIRST_LEVEL(Tree)

```
• procedure APPEND_TRAV(TestCaseOrStep, Tree, Level)
  begin
    if (Am in Level is not '+' TestCaseOrStep.DefaultReference) then
      begin (* Do only if Defaults not appended yet *)
        for (every alternative Ai in Level) do
          begin
            if (Ai is not a leaf) then
              begin
                NextLevel:=NEXT_LEVEL(Ai);
                APPEND_TRAV(TestCaseOrStep, Tree, NextLevel)
              end
            if (Ai is an ATTACH construct) then
              if (Ai.AttachedTree is TestStepIdentifier) then
                begin
                  NextTestStep:=Ai.AttachedTree;
                  NextTree:=ROOT_TREE(NextTestStep);
                  NextLevel:=FIRST_LEVEL(NextTree);
```

```
                    APPEND_TRAV(NextStep, NextTree, NextLevel);
                end
            else (* Attached tree is a TreeIdentifier *)
                begin
                    NextTestTree:=A_i.AttachedTree;
                    NextLevel:=FIRST_LEVEL(NextTree);
                    APPEND_TRAV(TestCaseOrStep, NextTree, NextLevel)
                end
        end
    APPEND_DEFAULT(TestCaseOrStep, NextTree, NextLevel)
  end
end
```

B.4.3 Removal of REPEAT constructs

If *TreeAndParameters* denotes a particular *TreeIdentifier* followed by an *ActualPARlist*, and condition denotes a particular Boolean expression, then: REPEAT *TreeAndParameters* UNTIL [*Condition]* can be replaced by:

```
          [TRUE]
label_A     [TRUE]
              + TreeAndParameters
                [NOT (condition)]
                    -> label_A
                [condition]
                    :
```

Lines describing subsequent behaviour of the REPEAT construct follow after this expansion, with an additional indentation of two levels.

B.4.4 Expanding ATTACHED trees

Attached trees are expanded by replacing the attach construct *+ TestStep* with the tree *TestStep*, and subsequently if there was behaviour specified following, and indented from, the ATTACH construct to insert this behaviour after, and indented from, each leaf in the attached tree.

When evaluating the semantics of a behaviour tree that has had Defaults appended and the REPEAT construct removed, it is recommended to expand attached trees on each level (set of alternatives) as that level is reached in executing the Test Case. The attached trees on Level are expanded using the following procedure:

-
```
procedure EXPAND_LEVEL(Level)
begin
    for (every alternative A_i in Level) do
        begin
            if (A_i is an ATTACH construct) then
                begin
                    Subsequent:=SUBSEQUENT_BEHAVIOUR_TO(A_i);
                    EXPAND(A_i.AttachedTree, Subsequent, A_i, Leve)
                end
        end
end
```

-
```
procedure EXPAND(SubTree, Subsequent, Alternative, Level)
begin
    REPLACE_PARAMETERS(Alternative, Subtree);
    (* This replaces the formal parameters in Subtree by the formal parameters specified in the actual parameter list of Alternative,
       doing so by textual substitution *)
    for (every leaf L_i in Subtree) do APPEND_SUBSEQUENT_BEHAVIOUR_TO(L_i, Subsequent);
```

```
    New_Level:=FIRST_LEVEL(Subtree);
    Level:=REPLACE(Level, New_Level, Alternative)
    (* This replaces Alternative in Level with the array New_level, e.g., REPLACE((A,B,C), (X,Y,Z), B) gives: (A,X,Y,Z,C) *)
end
```

The expansion of attached trees is also explained in 14.13.

B.5 TTCN operational semantics

B.5.1 Introduction

It is assumed that all sub-trees (both direct and indirect attachments) and REPEAT trees, have been previously expanded, according to the rules defined in the main body of this standard. It is also assumed that all Defaults have been appended according to the rules defined in B.4.2.

An event, pseudo-event, or construct in TTCN is considered to *match* when it can be successfully evaluated. 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 Introduction to the pseudo-code notation

TTCN semantics are defined using a simple functional approach that explains the execution of a TTCN behaviour tree and the components that form the nodes in that 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.

The functions assume the existence of two variables (these have nothing to do with TTCN variables) they are:

SendObject	is a temporary global data structure whose value is an ASP or PDU that is to be sent. This value is constructed according to the relevant constraint specifications.
ReceivedObject	is a temporary global data structure whose value is a copy of an ASP or PDU that has been received.

Execution of a TTCN Test Case begins with invoking EVALUATE_TEST_CASE.

B.5.3 Execution of a test case

B.5.3.1 Execution of a Test Case - pseudo-code

- **function EVALUATE_TEST_CASE**(BehaviourTree) :**BOOLEAN**

```
    (* Level is a variable local to EVALUATE_TEST_CASE. If A_i is an alternative in a set of m alternatives in the current behaviour
       tree then Level is the ordered set (A_1, A_2 . . . . . A_m) where each A_i is either an event, a pseudo-event or a construct. Note
       that because the tree is fully expanded the only TTCN statement that can appear is the GOTO *)
begin
    Level:= FIRST_LEVEL;
    EVALUATE_LEVEL (Level)
end
```

- **function EVALUATE_LEVEL** (Level) :**BOOLEAN**

```
    (* The alternatives A_1 . . . . . A_m contained in level are processed in their order of appearance. TTCN operational semantics
       assume that the processing of a set of alternatives is instantaneous i.e., the status of any of the events cannot change during
       the matching process. Note that level is updated to next level by the appropriate statement function *)
begin
    repeat
        TAKE_SNAPSHOT;
        if EVALUATE_EVENT_LINE(A_1, Level) then EVALUATE_LEVEL(Level);
        if EVALUATE_EVENT_LINE(A_2, Level ) then EVALUATE_LEVEL(Level);
        if          ...
```

...
...
if EVALUATE_EVENT_LINE(A_m, Level) **then** EVALUATE_LEVEL(Level)
until SNAPSHOT_FIXED(Level)
RETURN(TestCaseError)
(* where SNAPSHOT_FIXED(Level) returns TRUE if all relevant PCO queue(s) have some event(s) on them and all relevant timers have expired, and otherwise FALSE. *)
end

- **function EVALUATE_EVENT_LINE(** A_i,
Level) :**BOOLEAN**
(* This function calls EVALUATE_EVENT, EVALUATE_PSEUDO_EVENT or EVALUATE_CONSTRUCT depending on what type of 'event' the current alternative (A_i) is *)
case A_i **of**
EVENT: **if** EVALUATE_EVENT (A_i, Level) **then** RETURN(TRUE) **else** RETURN(FALSE);
PSEUDO-EVENT: **if** EVALUATE_PSEUDO_EVENT (A_i, Level) **then** RETURN(TRUE)
else RETURN(FALSE);
TTCN_CONSTRUCT: **if** EVALUATE_CONSTRUCT (A_i, Level) **then** RETURN(TRUE)
else RETURN(FALSE)
end

B.5.3.2 Execution of a Test Case - natural language description

Step 1. Test Case evaluation begins at the leftmost level of indentation (that is, the lines which are not yet indented in the TTCN.GR tree) of the tree.

Step 2. A snapshot of the incoming PCO queue(s) and timeout list is taken.

NOTE 1 - The act of taking a snapshot does not remove an event from the PCO.

Within the behaviour line at the current level of alternatives, consider the first one specified.

Step 3. Evaluate the TTCN statement on the current behaviour line, based on what is specified in the leftmost element, except that those TTCN statements that begin with a PCO identifier, are evaluated based on what follows the PCO identifier.

The evaluation of each type of TTCN statement is specified in the operational semantics for that TTCN statement type. Assignments, timer operations and SEND events are considered to evaluate to a *successful match* unless qualified by a Boolean expression that evaluates to False. IMPLICIT SEND and GOTO statements are always considered as successful matches. The RECEIVE event may or may not match, depending on both the Boolean expression, if any, and the first event on the relevant PCO queue.

Step 4. If the TTCN statement evaluates to a successful match, then go to Step 5.

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 3.

If there are no more alternatives and yet all PCO 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 there is a test case error and the Test Case shall be stopped indicating *test case error*.

NOTE 2 - This is a test case error because under these conditions none of the set of alternatives can ever match.

In all other cases, take a new snapshot of the PCO queue(s) and timeout list and consider again the first TTCN statement in the set of alternatives; then go to Step 3.

Step 5. If a leaf node in the tree has been reached, then go to Step 6.

Otherwise, consider a new set of alternatives for evaluation. (This set consists of the TTCN statements immediately following, and indented a single level from, the TTCN statement that has just matched.). Go to Step 2.

Step 6. A leaf node of the tree is reached, use the current value of the preliminary result variable R as the final verdict of the Test Case as in B.5.16.2.

B.5.4 Functions for TTCN events

B.5.4.1 Functions for TTCN events - pseudo-code

- **function EVALUATE_EVENT**(A_i, Level) :**BOOLEAN**

```
(* This function calls SEND, IMPLICIT SEND, RECEIVE, OTHERWISE or TIMEOUT depending on what type of event the current
   alternative (Ai) is *)

case Ai of
   SEND :          if SEND (Ai, Level) then RETURN(TRUE) else RETURN(FALSE);
   RECEIVE:        if RECEIVE (Ai, Level) then RETURN(TRUE) else RETURN(FALSE);
   OTHERWISE:      if OTHERWISE (Ai, Level) then RETURN(TRUE) else RETURN(FALSE);
   TIMEOUT:        if TIMEOUT (Ai, Level) then RETURN(TRUE) else RETURN(FALSE);
   IMPLICIT_SEND:  if IMPLICIT_SEND (Level) then RETURN(TRUE)
end
```

B.5.4.2 Functions for TTCN events - natural language description

If the TTCN statement is an event, then it will be evaluated as specified in B.5.5.2 for a SEND event; B.5.6.2 for a RECEIVE event; B.5.7.2 for an OTHERWISE event; B.5.8.2 for a TIMEOUT event; or B.5.9.2 for an IMPLICIT SEND event.

B.5.5 Execution of the SEND event

B.5.5.1 Execution of the SEND event - pseudo-code

- **function SEND** (PCOidentifier,
  ASPidentifier or PDUidentifier,
  Qualifier,
  Assignment,
  TimerOperation,
  ConstraintsReference,
  Verdict,
  Level) :**BOOLEAN**

```
(* All parameters except Level are picked up from Ai *)

begin
   if EVALUATE_BOOLEAN (Qualifier) then
      begin
         BUILD_SEND_OBJECT (ASPidentifierOrPDUidentifier, ConstraintsReference);
         EXECUTE_ASSIGNMENT (Assignment)
         SEND_EVENT (PCOidentifier )
         TIMER_OP (TimerOperation);
         VERDICT(Verdict);
         Level:= NEXT_LEVEL;
         LOG(PCOidentifier, SendObject);
         RETURN(TRUE)
      end
   else RETURN (FALSE)
end
```

- **function BUILD_SEND_OBJECT** (ASPidentifierOrPDUidentifier,
  ConstraintsReference) :**BOOLEAN**

```
begin
   SendObject :=  ( an instance of ASPidentifierOrPDUidentifier
                    whose parameters/fields have the values specified by ConstraintsReference);
```

```
    RETURN(TRUE)
end
```

B.5.5.2 Execution of the SEND event - natural language description

The contents of the ASP or PDU, 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 be executed.
- If the qualifier evaluates to TRUE, then continue with Step 2.

Step 2. The ASP or PDU declaration will be assigned the values specified in the named Constraints Reference.

Step 3. 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.

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 being built for comparison. The structure defined for the associated Constraints Reference is used for this named parameter or field.

Step 4. If there is an Assignment statement, then that assignment will be performed as in B.5.12.2.

Step 5. The ASP or PDU is now fully filled in according to the specifications given. The LT or UT will send the ASP or PDU. (If a PCO was stated, the ASP or PDU is to be sent at that PCO. 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.)

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.5.13.

Step 7. If a verdict is coded, process the verdict as in B.5.16.2.

Step 8. Record in the conformance log the following information, as well as the information specified in B.5.17.2:

- the PCO at which the SEND occurred;
- the fully defined ASP, PDU or TCP that was sent.

B.5.6 Execution of the RECEIVE event

B.5.6.1 Execution of the RECEIVE event - pseudo-code

```
• function RECEIVE(     PCOidentifier,
                        ASPidentifier or PDUidentifier,
                        Qualifier,
                        Assignment,
                        TimerOperation
                        ConstraintsReference,
                        Verdict,
                        Level) :BOOLEAN
    (* All parameters except Level are picked up from Ai *)

  begin
    if  RECEIVE_EVENT (PCOidentifier) then
        begin
          if (    RECEIVED_OBJECT(ASPidentifierOrPDUidentifier, ConstraintsReference)
                  AND EVALUATE_BOOLEAN (Qualifier) )
              then
                  begin
                      EXECUTE_ASSIGNMENT (Assignment);
                      TIMER_OP (TimerOperation);
                      REMOVE_OBJECT (PCOidentifier);
                      VERDICT(Verdict);
                      Level:=NEXT_LEVEL;
                      LOG(PCOidentifier, ReceivedObject);
```

```
                    RETURN(TRUE)
                end
            else RETURN (FALSE)
        end
    else RETURN (FALSE)
end
```

- **function RECEIVE_EVENT** (PCOidentifier) :**BOOLEAN**

```
    (* The actual object is NOT removed from the PCOidentifier queue *)
begin
    if INPUT_Q (PCOidentifier) NOT empty then
        begin
            ReceivedObject := copy of object at head of INPUT_Q(PCOidentifier);
            RETURN(TRUE)
        end
    else RETURN (FALSE)
end
```

- **function RECEIVED_OBJECT** (ASPidentifierOrPDUidentifier,
  ConstraintsReference) :**BOOLEAN**

```
begin
    if (  (ReceivedObject is ASPidentifierOrPDUidentifier)
          AND
          (parameters/fields of ReceivedObject have the values specified by the ConstraintsReference) )
        then RETURN(TRUE)
    else RETURN (FALSE)
end
```

B.5.6.2 Execution of the RECEIVE event - natural language description

The LT or UT will check to see if the contents of the ASP or PDU, as specified in the named Constraints Reference entry, have been received. Note that if there is a qualifier, the RECEIVE can only match if that qualifier evaluates to TRUE.

Step 1. If the snapshot that was taken when beginning the current iteration of checking this level of alternatives for matching shows that there *no* incoming ASP or PDU, then this RECEIVE cannot match.

Otherwise, continue to Step 2.

Step 2. A copy of the ASP or PDU is assembled, using the structure defined in the ASP or PDU declaration plus the values specified in the named Constraints Reference. This copy will be used for comparison against the incoming ASP or PDU (if any), to determine if the RECEIVE can match as specified.

Step 3. 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 used in the comparison.

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 being built for comparison. The structure defined for the associated Constraints Reference is used for this named parameter or field.

Step 4. The ASP or PDU is now fully filled in according to the specifications given. The LT or UT will compare the copy created from the specified data to the ASP or PDU that has been received in the snapshot, if any.

If a PCO was stated, the ASP or PDU shall have been received at that PCO. 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.

Step 5. If there is a qualifier, then that qualifier will be evaluated.

- If the qualifier evaluates to FALSE, the RECEIVE cannot match.
- If the qualifier evaluates to TRUE, then continue with Step 6.

Step 6. If the LT or UT is unable to match the ASP or PDU as specified (***e.g.,*** no ASP or PDU has arrived, an ASP

or PDU has arrived but it does not match the data as specified, or the ASP or PDU matches but the qualifier does not hold), then this RECEIVE event is to be considered as not having matched, *i.e.,* the next alternative to this RECEIVE will be attempted.

If the RECEIVE did match successfully, continue to Step 7.(The incoming ASP or PDU which has just matched will be removed from the incoming PCO queue, and will therefore not be available for matching against any subsequent events in the Test Case.)

Step 7. If there is an Assignment statement, then that assignment will be performed as in B.5.12.2.

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.5.13.

Step 9. If a verdict is coded, process the verdict as in B.5.16.2.

Step 10. Record in the conformance log the following information, as well as the information specified in B.5.17.2:

- the PCO at which the RECEIVE occurred;
- the fully defined ASP, PDU or TCP that was received.

B.5.7 Execution of the OTHERWISE event

B.5.7.1 Execution of the OTHERWISE event - pseudo-code

```
• function OTHERWISE (      PCOidentifier,
                            Qualifier,
                            Assignment,
                            TimerOperation,
                            Verdict,
                            Level) :BOOLEAN
    (* All parameters except Level are picked up from Ai *)

  begin
    if  (   (RECEIVE_EVENT (PCOidentifier))
            AND  (EVALUATE_BOOLEAN (Qualifier) )
        then
            begin
                EXECUTE_ASSIGNMENT (Assignment);
                TIMER_OP (TimerOperation);
                REMOVE_OBJECT (PCOidentifier);
                VERDICT(Verdict);
                Level:= NEXT_LEVEL;
                LOG(PCOidentifier, ReceivedObject);
                RETURN (TRUE)
            end
      else RETURN (FALSE)
  end
```

B.5.7.2 Execution of the OTHERWISE event - natural language description

The tester shall accept any incoming data 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 a PCO was stated, the ASP or PDU shall have been received at that PCO. 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.

Step 2. If there is a qualifier, then that qualifier will be evaluated after other processing takes place.

- If the qualifier evaluates to FALSE, the OTHERWISE cannot match.
- If the qualifier evaluates to TRUE, then continue with Step 3.

Step 3. If the tester is unable to receive an ASP or PDU (*i.e.,* no ASP or PDU has arrived), then this OTHERWISE event is to be considered as not having matched, and the next alternative to this OTHERWISE will be attempted.

If the OTHERWISE did take place successfully, continue to Step 4. (The incoming ASP or PDU which has just matched will not be available for matching against any subsequent events in the Test Case.)

Step 4. If there is an Assignment statement, then that assignment will be performed as in B.5.12.2.

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.5.13.

Step 6. If a verdict is coded, process the verdict as in B.5.16.2.

Step 7. Record in the conformance log the following information, as well as the information specified in B.5.17.2:

- the PCO at which the OTHERWISE occurred;
- the fully defined ASP, PDU that was received.

B.5.8 Execution of the TIMEOUT event

B.5.8.1 Execution of the TIMEOUT event - pseudo-code

```
function TIMEOUT (      TimerIdentifier,
                        Qualifier,
                        Assignment,
                        TimerOperation,
                        Verdict,
                        Level) :BOOLEAN
    (* All parameters except Level are picked up from Ai *)

begin
    if EVALUATE_BOOLEAN (Qualifier) then
        begin
            if (TIMER_EXPIRED (TimerIdentifier)) then
                begin
                    EXECUTE_ASSIGNMENT (Assignment);
                    TIMER_OP (TimerOperation);
                    VERDICT(Verdict);
                    Level:= NEXT_LEVEL;
                    LOG(Timerlidentifier);
                    RETURN(TRUE);
                end
            else RETURN (FALSE)
        end
    else RETURN (FALSE)
end

function TIMER_EXPIRED (TimerIdentifier) :BOOLEAN
begin
    if (timer has expired) then
        begin
            reset expired timer;          (* see B.5.8.2 *)
            RETURN (TRUE)
        end
    else
        RETURN (FALSE)
end
```

B.5.8.2 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.5.12.2.

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.5.13.

Step 5. If a verdict is coded, process the verdict as in B.5.16.2.

Step 10. Record in the conformance log the following information, as well as the information specified in B.5.17.2:

- the name of the timer that expired.

B.5.9 Execution of the IMPLICIT SEND event

B.5.9.1 Execution of the IMPLICIT SEND event - pseudo-code

- **function IMPLICIT_SEND** (Level) :**BOOLEAN**

```
begin
    (* Evaluate IMPLICIT_SEND according to 14.9.6 * *)
    Level:= NEXT_LEVEL;
    RETURN(TRUE)
end
```

B.5.9.2 *Execution of* IMPLICIT SEND - *natural language description*

The IUT does whatever is necessary to send the contents of the ASP or PDU, as specified in the named constraints reference entry.

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.

B.5.10 Execution of the PSEUDO-EVENT

B.5.10.1 Execution of PSEUDO-EVENTS - pseudo-code

- **function EVALUATE_PSEUDO_EVENT** (Qualifier,
  Assignment,
  TimerOperation,
  Verdict,
  Level) :**BOOLEAN**

```
    (* All parameters except Level are picked up from Ai *)

begin
    if EVALUATE_BOOLEAN (Qualifier)
        then
            begin
                ASSIGNMENT (Assignment);
                TIMER_OP (TimerOperation);
                VERDICT(Verdict);
                Level:= NEXT_LEVEL;
                LOG( );
```

```
                RETURN(TRUE)
            end
        else RETURN (FALSE)
    end
```

B.5.10.2 Execution of PSEUDO-EVENTS - natural language description

If the TTCN statement is a pseudo-event, then it will be evaluated as specified in B.5.11.2 for a Boolean Expression; B.5.12.2 for an Assignment Statement; B.5.13.2 for a START timer pseudo-event; B.5.13.3 for a CANCEL timer pseudo-event; or B.5.13.4 for a READ timer pseudo-event.

Step n. After completion of the pseudo event record in the conformance log the information specified in B.5.17.2:

B.5.11 Execution of BOOLEAN expressions

B.5.11.1 Execution of BOOLEAN expressions - pseudo-code

- **function EVALUATE_BOOLEAN**(Qualifier) :**BOOLEAN**
- (* for further definition see B.5.11.2 *)

```
begin
    if no argument then RETURN (TRUE)
    else   begin
               if  Qualifier  then RETURN (TRUE)
               else RETURN (FALSE)
           end
end
```

B.5.11.2 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.

Step 2. If the condition expressed by the Boolean expression evaluates to FALSE, then this Boolean expression is not a matching event. Skip the remaining steps of this clause.

If the Boolean expression has been evaluated and determined to match, the subsequent processing depends upon whether the Boolean expression was coded as part of an event (*i.e.,* SEND, RECEIVE, TIMEOUT, or OTHERWISE), or whether it was coded as a pseudo-event.

If the Boolean expression was coded with an event, then this segment of the statement line is considered to match, and the remainder of the statement line will be evaluated according to the semantics for that event type. Skip the following steps, as they apply only to Boolean expressions coded as pseudo-events.

If this Boolean expression was coded as a pseudo-event, then this Boolean is considered as a matching behaviour. Proceed to the next step to process the rest of the Boolean expression behaviour line.

Step 3. If there is an Assignment statement, then that assignment will be performed as in B.5.12.2.

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.5.13.

Step 5. If a verdict is coded, process the verdict as in B.5.16.2.

B.5.12 Execution of ASSIGNMENTs

B.5.12.1 Execution of EXECUTE_ASSIGNMENT - pseudo-code

- **function EXECUTE_ASSIGNMENT** (Assignment) :**BOOLEAN**

(* note that assignments of the kind *<PDUidentifier> . <FIELDidentifier>* etc. assign (reassign) values to fields/ parameters in SendObject. This kind of assignment shall not be used with a RECEIVE event *)

```
begin
    if no argument then RETURN (TRUE)
    else   begin
               execute the clauses in Assignment in a left-to-right order;
               RETURN(TRUE)
           end
end
```

B.5.12.2 Execution of ASSIGNMENTs - natural language description

An assignment statement specifies that the variable on the left-hand side of that statement is to take on the value of the right-hand side of the statement. An assignment statement may be stated as part of a statement line (*i.e.,* on the same line with a SEND, RECEIVE, TIMEOUT, or OTHERWISE), in combination with a Boolean expression, or as a behaviour line on its own (*i.e.,* as a pseudo-event).

Step 1. The manner in which the assignment is to be performed depends upon the format used in specifying the right-hand side of the assignment. The clauses are evaluated in left to right order, observing the precedence indicated in table 3.

Step 2. Once the assignment has been performed, the subsequent processing depends upon whether the assignment statement was coded as part of an event (*i.e.,* SEND, RECEIVE, TIMEOUT, or OTHERWISE), or whether it was coded as a pseudo-event.

If this assignment statement was coded as a pseudo-event, then proceed to the next step to process the rest of the assignment statement behaviour line.

Step 3. If one or more timer operations were coded on the behaviour line, the appropriate timer operation(s) will be performed as in B.5.13.

Step 4. If a verdict is coded, process the verdict as in B.5.16.2.

B.5.13 Execution of TIMER operations

B.5.13.1 Execution of TIMER operations - pseudo-code

```
• function TIMER_OP (TimerOperation) :BOOLEAN
  begin
      if no argument then RETURN (TRUE)
      else   case TimerOperation  of
                 START_TIMER:      if   START_TIMER(TimerOperation) then RETURN(TRUE)
                                   else RETURN(FALSE);        (* see B.5.13.2 for START_TIMER *)
                 CANCEL_TIMER:     if   CANCEL_TIMER(TimerOperation) then RETURN(TRUE)
                                   else RETURN(FALSE);        (* see B.5.13.3 CANCEL_TIMER *)
                 READ_TIMER:       if   READ_TIMER(TimerOperation) then RETURN(TRUE)
                                   else RETURN(FALSE);        (* see B.5.13.4 READ_TIMER *)
             end
  end
```

B.5.13.2 Execution of START timer - natural language description

The START timer operation specifies that a timer is to begin ticking. This operation may be coded as part of a statement line (*i.e.,* on the same line with a SEND, RECEIVE, OTHERWISE or TIMEOUT), or as a behaviour line on its own, or in combination with a qualifier and/or assignment statement.

Step 1. Determine the duration to be used for this instance of this timer. If no duration is specified on this operation (as an integer value or expression), the default duration from the timer declarations will be used.

If a duration has been specified, however, this duration will be used instead of the default that was given in the timer declaration. Overriding a default timer duration by coding a duration on the START timer operation applies only to this instance of the timer - if this timer is started at any other point in the test it is not affected by this override of the duration.

Step 2. If this timer is already running, it is to be cancelled and restarted, so that no time has elapsed. If the timer is not yet running, it 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.

Step 3. If another timer operation is coded following this START, then process that timer operation (exclusive of verdict processing).

Step 4. If a verdict is coded, process the verdict as in B.5.16.2.

B.5.13.3 CANCEL timer - natural language description

The CANCEL timer operation specifies that a timer (or timers) is to stop ticking. This operation may be coded as part of a statement line (*i.e.,* on the same line with a SEND, RECEIVE, OTHERWISE or TIMEOUT), or as a behaviour line on its own, or in combination with a qualifier and/or assignment statement.

Step 1. Determine the name of the timer(s) to be cancelled.

- if no timer identifier is specified, then the LT or UT shall cancel *all* timers that had been running.
- if a timer identifier is specified, then the LT or UT shall cancel this timer (which had been running.

Step 2. The status of the named or implied timer(s) is to be changed to "not running". The amount of time elapsed for the timer (or timers) 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.

Step 3. If another timer operation is coded following this CANCEL, then process that timer operation (exclusive of verdict processing).

Step 4. If a verdict is coded, process the verdict as in B.5.16.2.

B.5.13.4 READTIMER - natural language description

The READTIMER 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. This operation may be coded as part of a statement line (*i.e.,* on the same line with a SEND, RECEIVE, OTHERWISE or TIMEOUT), or as a behaviour line on its own, or in combination with a qualifier and/or assignment statement.

Step 1. Interrogate the value of the timer having the specified name. Store the amount of time passed into the named variable. The units returned are the same as the units declared for this timer type.

If the timer is not currently running, the named variable shall be set to zero.

Step 2. If another timer operation is coded following this READTIMER, then process that timer operation (exclusive of verdict processing).

Step 3. If a verdict is coded, process the verdict as in B.5.16.2.

B.5.14 Functions for TTCN constructs

B.5.14.1 Functions for TTCN constructs - pseudo-code

- **function EVALUATE_CONSTRUCT** (Construct, Level) :**BOOLEAN**

```
    (* All parameters except Level are picked up from Ai *)

    (* because the Test Case tree is fully expanded prior to processing the REPEAT and ATTACH constructs are never encountered,
       see B.4.3 and B.4.4 *)

  case Construct of
    GOTO:    GOTO(Label, Level);
    RETURN (TRUE)
  end
```

B.5.14.2 Functions for TTCN constructs - natural language description

If the TTCN statement is a TTCN construct, then it will be evaluated as specified in B.5.15.2 for a GOTO construct; see clause B.4 for a description of REPEAT and ATTACH constructs.

B.5.15 Execution of the GOTO construct

B.5.15.1 Execution of the GOTO construct - pseudo-code

```
• function GOTO (Label, Level) :BOOLEAN
  begin
     Level:= LABELED_LEVEL(Label);
     RETURN (TRUE)
  end
```

B.5.15.2 Execution of the GOTO construct - natural language description

The LT or UT shall cause control to transfer from the current event (*i.e.*, the GOTO construct) to the set of alternatives having the specified target label in the labels column. Execution now continues at this new level.

B.5.16 The VERDICT

B.5.16.1 The VERDICT - pseudo-code

```
• function VERDICT(VerdictColumnEntry) :BOOLEAN
  begin
     if VerdictColumnentry is blank then RETURN(TRUE)
     else   begin
               if VerdictColumnEntry = R then LOG(Verdict implied by R);
               else LOG(VerdictColumn);
               if   VerdictColumnEntry is preliminary result then (* Contains verdict *)
                  begin
                     if  ((R = none) or (R=pass and VerdictColumnEntry <> (PASS))
                        or (R=inconc and VerdictColumnEntry = (FAIL)) then
                        begin
                           if VerdictColumnEntry = (PASS) then R := pass;
                           if VerdictColumnEntry = (INCONC) then R :=inconc;
                           if VerdictColumnEntry = (FAIL) then R := fail;
                        end
                  end
               else   (* Final verdict *)
                  begin
                     reset test case variables all timers;
                     if ( ( VerdictColumnEntry = R and R = none ) or
                        ( VerdictColumnEntry = PASS and R = inconc ) or
                        ( VerdictColumnEntry = PASS and R = fail ) or
                        ( VerdictColumnEntry = INCONC and R = fail ))
                     then raise test case error;
                     STOP (* Test Case terminates here *)
                  end
            end
  end
```

B.5.16.2 The VERDICT - natural language description

If a verdict is coded, process the verdict.

- If the verdict is enclosed in parentheses, then the temporary result variable R will be updated according to the verdict algorithm in 15.17.2. The stated verdict is recorded in the conformance log.
- If the verdict is R, then the current value of the temporary result variable 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 the stated verdict will be used as the final verdict for the Test Case. If the final verdict is inconsistent with the preliminary verdict, raise a *TestCaseError*.

B.5.17 The Conformance Log

B.5.17.1 The LOG - pseudo-code

- **procedure LOG (Variable number of arguments)**
  begin
  log the sequence 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 the verdict or preliminary result associated with the event line (if any);
  log current time; (* current time may be actual or relative *)
  end

B.5.17.2 The conformance log - natural language description

Record the following information in the conformance log:

- the sequence number of the event line (if any);
- the label associated with the event line (if any);
- the assignment(s) made (if any);
- the timer operation(s) performed (if any);
- the verdict or preliminary result associated with the event line (if any);
- time stamp;

B.5.18 Other miscellaneous functions used by the pseudo-code

- **function FIRST_LEVEL :LEVEL**
  begin
  RETURN(**set of alternatives at first level of indentation**)
  end

- **function NEXT_LEVEL :LEVEL**
  begin
  if (set of alternatives at next level of indentation exist) then
  RETURN (set of alternatives at next level of indentation)
  else
  VERDICT(R)
  STOP (* Test Case terminates here *)
  end

- **function LABELED_LEVEL** (Label) :**LEVEL**
  begin
  RETURN(**set of alternatives at the level of indentation indicated by** Label)
  end

- **function RETURN**(argument) :**BOOLEAN or LEVEL or QUEUE**
  exit the current function immediately and return the value of argument
  end

- **function OUTPUT_Q**(PCOidentifier) :**QUEUE**
  (* In TTCN each PCO is modeled as two unbounded FIFO queues: one INPUT queue (to the LT or UT) and one OUTPUT queue (from the LT or UT). In test suites that use only one PCO and when this PCO is not explicitly associated with an event then the single (default) PCO is picked up from the PCO declarations*)

begin
if no argument then RETURN(**default PCO output queue**)
else RETURN(**output queue identified by** PCOidentifier)
end

- **function INPUT_Q**(*PCOidentifier*) :**QUEUE**

  (* In TTCN each PCO is modeled as two unbounded FIFO queues: one INPUT queue (to theLT or UT) and one OUTPUT queue (from the LT or UT). In test suites that use only one PCO and when this PCO is not explicitly associated with an event then the single (default) PCO is picked up from the PCO declarations*)

begin
if no argument then RETURN(**default PCO input queue**)
else RETURN(**input queue identified by** PCOidentifier)
end

- **procedure TAKE_SNAPSHOT**

  (* Snapshot semantics are used for RECEIVE 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 *)

begin
update PCO input queues;
update TIMER queue;
end

- **procedure STOP**

begin
reset test case variables;
reset PCO queue;
reset TIMER queue;
reset timers;
terminate execution immediately
end

Annex C
(normative)

Compact proformas

C.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.

C.2 Compact proformas for constraints

C.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; and

b) 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.

C.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:

ASP Constraints Declarations					
ASP Type : *ASP_Identifier*					
Constraint Name	**Derivation Path**	**Parameter 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 C.1 - (Compact) ASP Constraints Declarations

This proforma is used for ASPs and their parameters in the same way that PDU Constraints Declarations proforma is used for PDUs and their fields (see C.2.3).

C.2.3 Compact proformas for PDU constraints

C.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:

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 C.2 - (Compact) PDU Constraints Declarations

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 12.6). 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 as 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.

EXAMPLE C.1 - Constraints using the compact constraints proforma

C.1.1 Given the declaration of PDU_B to be

PDU Type Definition		
PDU Name : PDU_B **PCO Type** : XSAP **Comment** :		
Field Name	**Field Type**	**Comments**
FIELD1 FIELD2 FIELD3	INTEGER BOOLEAN IA5String	

C.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]

EXAMPLE C.2 - 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	FIELD4	
CN0		0	'FF'H	'00'B	TRUE	
CN1	CN0.	1				
CN2	CN0.CN.		-	?		

C.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 C.3 - 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	

C.2.4 Compact proformas for Structured Type constraints

Compact Structured Type constraints shall be provided in the following proforma:

Structured Type Constraints Declarations					
Structure 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 C.3 - (Compact) Structured Type Constraints Declarations

EXAMPLE C.4 - 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.

C.4.1 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	

C.4.2 A1 is a base constraint of Structured Type A:

Structured Type Constraint Declaration		
Constraint Name : A1 **Structured Type** : A **Derivation Path** : **Comment** :		
Element Name	**Element Value**	**Comments**
Y1 Y2 Y3	0 0 0	

C.4.3 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. **Comment** :		
Element Name	**Element Value**	**Comments**
Y2	3	

C.4.4 Structured Type A's constraints A1 and A2 in the compact form

Structured Type Constraints Declarations					
Structured Type Name: A					
Constraint Name	**Derivation Path**	**Element Name**			**Comments**
		Y1	Y2	Y3	
A1		0	0	0	
A2	A1.		3		

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.

C.2.5 Compact proformas for ASN.1 constraints

The following proformas 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**
ConsId&ParList$_1$	*ConstraintValue&Attribute*$_1$
⋮	⋮
ConsId&ParList$_m$	*ConstraintValue&Attribute*$_m$

Proforma C.4 - (Compact) ASN.1 ASP Constraints Declarations

ASN.1 PDU Constraints Declarations	
PDU Type: *PDU_Identifier*	
Constraint name	**ASN.1 Value**
ConsId&ParList$_1$	*ConstraintValue&Attributes*$_1$
⋮	⋮
ConsId&ParList$_m$	*ConstraintValue&Attributes*$_m$

Proforma C.5 - (Compact) ASN.1 PDU Constraints Declarations

ASN.1 Type Constraints Declarations	
Type Name: *ASN1_TypeIdentifier*	
Constraint name	**ASN.1 Value**
ConsId&ParList$_1$	*ConstraintValue&Attributes*$_1$
⋮	⋮
ConsId&ParList$_m$	*ConstraintValue&Attributes*$_m$

Proforma C.6 - (Compact) ASN.1 Type Constraints Declarations

C.3 Compact proforma for Test Cases

C.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.

C.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:

Test Case Dynamic Behaviours			
Group : *TestGroupReference* **Default** : *DefaultReference*			
Test Case Name	**Purpose**	**Test Step Attachment**	**Comments**
TestCaseIdentifier	*FreeText*	*Attach*	*[FreeText]*

Proforma C.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 C.5 - 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**
OVERIDE1	Omit the overide parameter, when file exists.	+ OVERRIDE (FCRERQ_001, FCRERP_001)
OVERIDE2	Omit the overide parameter, when file does not exist.	+ OVERRIDE (FCRERQ_002, FCRERP_002)