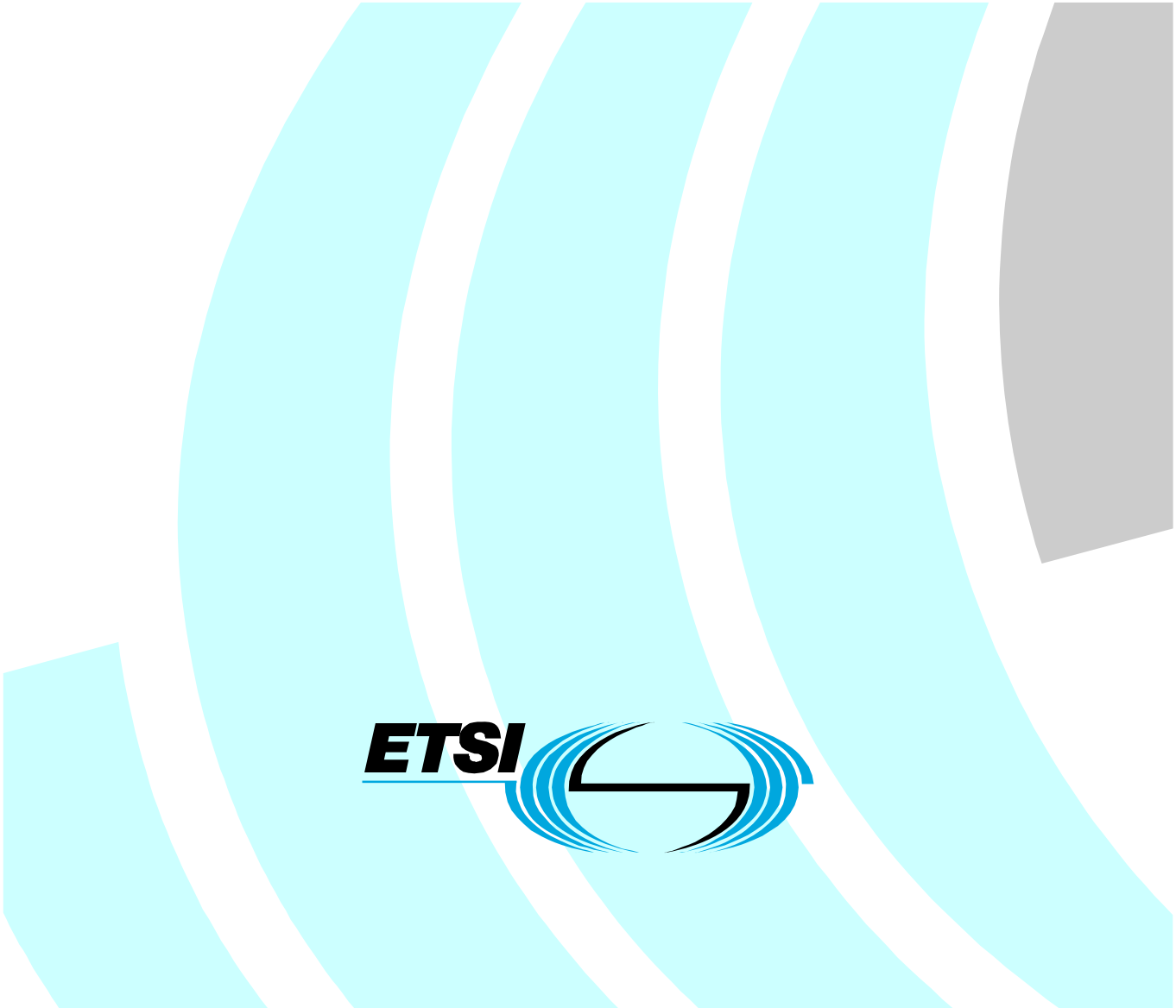
Final draft ETSI ES 201 873-1 V4.2.5 (2011-04)

ETSI Standard

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



Reference
RES/MTS-00107-1 T3 ed431 core

Keywords
methodology, MTS, testing, TTCN

ETSI

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

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

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

Important notice

Individual copies of the present document can be downloaded from:
http://www.etsi.org

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

Users of the present document should be aware that the document may be subject to revision or change of status. Information on the current status of this and other ETSI documents is available at
http://portal.etsi.org/tb/status/status.asp

If you find errors in the present document, please send your comment to one of the following services:
http://portal.etsi.org/chaircor/ETSI_support.asp

Copyright Notification

Contents

Intellectual Property Rights

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

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

Foreword

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

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

Part 1: "TTCN-3 Core Language";

Part 2: "TTCN-3 Tabular presentation Format (TFT)";

Part 3: "TTCN-3 Graphical presentation Format (GFT)";

Part 4: "TTCN-3 Operational Semantics";

Part 5: "TTCN-3 Runtime Interface (TRI)";

Part 6: "TTCN-3 Control Interface (TCI)";

Part 7: "Using ASN.1 with TTCN-3";

Part 8: "The IDL to TTCN-3 Mapping";

Part 9: "Use XML with TTCN-3";

Part 10: "TTCN-3 Documentation Comment Specification".

1 Scope

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

TTCN-3 is intended to be used for the specification of test suites which are independent of test methods, layers and protocols. Various presentation formats are defined for TTCN-3 such as a tabular presentation format (ES 201 873-2 [i.1]) and a graphical presentation format (ES 201 873-3 [i.2]). The specification of these formats is outside the scope of the present document.

While the design of TTCN-3 has taken the eventual implementation of TTCN-3 translators and compilers into consideration the means of realization of Executable Test Suites (ETS) from Abstract Test Suites (ATS) is outside the scope of the present document.

2 References

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the reference document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found at http://docbox.etsi.org/Reference.

NOTE: While any hyperlinks included in this clause were valid at the time of publication ETSI cannot guarantee their long term validity.

2.1 Normative references

The following referenced documents are necessary for the application of the present document.

[1] ETSI ES 201 873-4: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics".

[2] ISO/IEC 10646: "Information technology - Universal Multiple-Octet Coded Character Set (UCS)".

[3] ITU-T Recommendation X.292 Series X: "Data Networks and Open System Communications; Open Systems Interconnection - Conformance testing; OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications - The Tree and Tabular Combined Notation (TTCN)".

NOTE: The corresponding ISO/IEC standard is ISO/IEC 9646-3: "Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation (TTCN)".

[4] ITU-T Recommendation T.50: "Terminal Equipment and Protocols for Telematic Services; International Reference Alphabet (IRA) (Formerly International Alphabet No. 5 or IA5); Information technology - 7-Bit coded character set for information interchange".

NOTE: The corresponding ISO/IEC standard is ISO/IEC 646: "Information technology - ISO 7-bit coded character set for information interchange".

[5] ITU-T Recommendation X.290: "Data Networks and Open System Communications; Open Systems Interconnection - Conformance testing; OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications - General concepts".

NOTE: The corresponding ISO/IEC standard is ISO/IEC 9646-1: "Information technology - Open Systems Interconnection -Conformance testing methodology and framework; Part 1: General concepts".

[6] IEEE 754: "IEEE Standard for Floating-Point Arithmetic".

2.2 Informative references

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] ETSI ES 201 873-2: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 2: TTCN-3 Tabular presentation Format (TFT)".

[i.2] ETSI ES 201 873-3: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 3: TTCN-3 Graphical presentation Format (GFT)".

[i.3] ETSI ES 201 873-5: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI)".

[i.4] ETSI ES 201 873-6: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI)".

[i.5] ETSI ES 201 873-7: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 7: Using ASN.1 with TTCN-3".

[i.6] ETSI ES 201 873-8: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 8: The IDL to TTCN-3 Mapping".

[i.7] ETSI ES 201 873-9: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 9: Using XML schema with TTCN-3".

[i.8] ETSI ES 201 873-10: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 10: TTCN-3 Documentation Comment Specification".

[i.9] Void.

[i.10] Object Management Group (OMG) (2001): "The Common Object Request Broker: Architecture and Specification - IDL Syntax and Semantics". Version 2.6, FORMAL/01-12-01.

[i.11] ETSI ES 202 781: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Configuration and Deployment Support".

[i.12] ETSI ES 202 784: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Advanced Parameterization".

[i.13] ETSI ES 202 785: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Behaviour Types".

[i.14] ETSI ES 202 782: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: TTCN-3 Performance and Real Time Testing".

3 Definitions and abbreviations

3.1 Definitions

For the purposes of the present document, the terms and definitions given in ITU-T Recommendation X.290 [5], ITU-T Recommendation X.292 [3] and the following apply:

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

basic types: set of predefined TTCN-3 types described in clauses 6.1.0 and 6.1.1 of the present document

NOTE: Basic types are referenced by their names.

communication port: abstract mechanism facilitating communication between test components

NOTE: A communication port is modelled as a FIFO queue in the receiving direction. Ports can be message-based or procedure-based.

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

NOTE: Variables, constants, templates, etc. have compatible types if conditions in clause 6.3 are met.

completely initialized: values and templates of simple types are completely initialized if they are partially initialized

NOTE: Values and templates of structured types and arrays are completely initialized if all their fields and elements are completely initialized. In case of record of, set of, and array values and templates, this means at least the first n elements are initialized, where n is the minimal length imposed by the type length restriction or array definition (thus in case of n equals 0, the value "{}" also completely initializes a record of, a set of or an array).

data types: common name for simple basic types, basic string types, structured types, the special data type anytype and all user defined types based on them (see table 3 of the present document)

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

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

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

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

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

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

Implementation Conformance Statement (ICS): See ITU-T Recommendation X.290 [5].

Implementation eXtra Information for Testing (IXIT): See ITU-T Recommendation X.290 [5].

Implementation Under Test (IUT): See ITU-T Recommendation X.290 [5].

in parameterization: kind of parameterization where the value of the actual parameter (the argument) is bound to the formal parameter when the parameterized object is invoked, but the value of the formal parameter is not passed back to the actual parameter when the invoked object completes

NOTE 1: The arguments are evaluated before the parameterized object is entered.

NOTE 2: Only the values of the arguments are passed and changes to the arguments within the invoked object have no effect on the arguments as seen by the invoking object.

inout parameterization: kind of parameterization where the actual parameter is bound to the formal parameter when the parameterized object is invoked

NOTE 1: The invoked object uses the actual parameter directly, so that all changes made on the formal parameter become immediately effective on the actual parameter.

NOTE 2: Inout parameters can be used for functions, altsteps, and test cases only.

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

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

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

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

Main Test Component (MTC): See ITU-T Recommendation X.292 [3].

out parameterization: kind of parameterization where the value of the actual parameter (the argument) is not bound to the formal parameter when the parameterized object is invoked, but the value of the formal parameter is passed back to the actual parameter when the invoked object completes

NOTE 1: Out parameters can be used for functions, altsteps, and test cases only.

NOTE 2: An **`out`** formal parameter is uninitialized (unbound) when the invoked object is entered.

NOTE 3: The value is passed back to the actual parameter only if within the invoked object a value is assigned to it. If no value is assigned, the actual parameter remains unchanged when the invoked object completes.

Parallel Test Component (PTC): See ITU-T Recommendation X.292 [3].

partially initialized: values are partially initialized if a concrete value has been assigned to it or to at least one of its fields or elements

NOTE 1: A template variable is initialized if a matching mechanism has been assigned to it or to at least one of its fields or elements, directly or indirectly via expansion (see clause 15.6). A template is initialized if a matching mechanism has been assigned to it, directly or indirectly via expansion (see clause 15.6).

NOTE 2: Thus, constants and templates are always initialized at declaration. Variables (both value and template) are initialized if they, or at least one of their fields or elements has been used on the left hand side of an assignment (including initial value assignment at declaration). Module parameters are initialized either at declaration or by the test system before test execution.

port parameterization: ability to pass a port as an actual parameter into a parameterized object via a port parameter

NOTE: This actual port parameter is added to the specification of that object and may complete it.

qualified name: TTCN-3 elements can be identified unambiguously by qualified names

NOTE: For modules, the qualified name is the <module name>. For global definitions such as testcases, functions, etc., the qualified name is <module name>.<definition name>. For control, the qualified name is <module name>.control. For local definitions, such as variables, local templates, etc. within a global definition, the qualified name is <module name>.<global definition name>.<local definition name>.

right hand side (of assignment): expression, template reference or signature parameter identifier which stands right to an assignment symbol (:=)

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

root type: root types of types derived from TTCN-3 basic types are the respective basic types

NOTE 1: The root type of user defined record types is **`record`**, the root type of user defined record of and array types is **`record of`**, the root type of user defined set types is **`set`**, the root type of user defined set of types is **`set of`**. The root type of user defined union types is **union** and the root type of anytypes is **`anytype`**. The root types of special configuration types are **`default`** or **`component`**, respectively. Port types do not have a root type.

NOTE 2: As **`address`** is more a predefined type name than a distinct type with its own properties, the root type of an **`address`** type and all of its derivatives are the same, as the root type was, if the type was defined with a name different from **`address`**.

static parameterization: form of parameterization, in which actual parameters are independent of run-time events; i.e. known at compile time or in case of module parameters are known by the start of the test suite execution

NOTE 1: A static parameter is to be known from the test suite specification, (including imported definitions), or the test system is aware of its value before execution time.

NOTE 2: All types are known at compile time, i.e. are statically bound.

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

System Under Test (SUT): See ITU-T Recommendation X.290 [5].

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

template parameterization: ability to pass a template as an actual parameter into a parameterized object via a template parameter

NOTE 1: This actual template parameter is added to the specification of that object and may complete it.

NOTE 2: Values passed to template formal parameters are considered to be in-line templates (see clause 15.4).

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

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

test case: See ITU-T Recommendation X.290 [5].

test case error: See ITU-T Recommendation X.290 [5].

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

test system: See ITU-T Recommendation X.290 [5].

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

timer parameterization: ability to pass a timer as an actual parameter into a parameterized object via a timer parameter

NOTE: This actual timer parameter is added to the specification of that object and may complete it.

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

unqualified name: the unqualified name of a TTCN-3 element is its name without any qualification

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

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

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

NOTE: Values may be constants or variables.

value parameterization: ability to pass a value as an actual parameter into a parameterized object via a value parameter

NOTE: This actual value parameter is added to the specification of that object and may complete it.

3.2 Abbreviations

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

API	Application Programming Interface
ATS	Abstract Test Suite
BMP	Basic Multilingual Plane
BNF	Backus-Nauer Form
CORBA	Common Object Request Broker Architecture
ETS	Executable Test Suite
FIFO	First In First Out
ICS	Implementation Conformance Statement
IRV	International Reference Version
IUT	Implementation Under Test
IXIT	Implementation eXtra Information for Testing
MTC	Main Test Component
PTC	Parallel Test Component
SUT	System Under Test
TCI	TTCN-3 Control Interfaces
TRI	TTCN-3 Runtime Interfaces
TSI	Test System Interface
TTCN-3	Testing and Test Control Notation version 3

4 Introduction

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

TTCN-3 includes the following essential characteristics:

- the ability to specify dynamic concurrent testing configurations;
- operations for procedure-based and message-based communication;
- the ability to specify encoding information and other attributes (including user extensibility);

- the ability to specify data and signature templates with powerful matching mechanisms;
- value parameterization;
- the assignment and handling of test verdicts;
- test suite parameterization and test case selection mechanisms;
- combined use of TTCN-3 with other languages;
- well-defined syntax, interchange format and static semantics;
- different presentation formats (e.g. tabular and graphical presentation formats);
- a precise execution algorithm (operational semantics).

NOTE: The present document uses the following pattern of concept description: concepts, principles and mechanisms are explained in (introductory) text at the beginning of a clause. For every concept having concrete syntax, the syntactical structure of that concept is presented afterwards. The syntactical structure follows the conventions for the TTCN-3 syntax description in clause A.1.1 and uses rules of the TTCN-3 BNF given in clause A.1. A semantic description follows the syntactic structure. The restrictions on the concept are listed subsequently. Finally, examples on the usage of the concept are given.

In case of a contradiction between the body of the present document (clauses 5 to 27) and annex A of the present document, annex A has the priority.

4.1 The core language and presentation formats

The TTCN-3 specification is separated into several parts (see figure 1).

The first part, defined in the present document, is the core language.

The second part, defined in ES 201 873-2 [i.1], is the tabular presentation format.

The third part, defined in ES 201 873-3 [i.2], is the graphical presentation format.

The fourth part, ES 201 873-4 [1], contains the operational semantics of the language.

The fifth part, ES 201 873-5 [i.3], defines the TTCN-3 Runtime Interface (TRI).

The sixth part, ES 201 873-6 [i.4], defines the TTCN-3 Control Interfaces (TCI).

The seventh part, ES 201 873-7 [i.5], specifies the use of ASN.1 definitions with TTCN-3.

The eight part, ES 201 873-8 [i.6], specifies the use of IDL definitions with TTCN-3.

The ninth part, ES 201 873-9 [i.7] specifies the use of XML definitions with TTCN-3.

The tenth part, ES 201 873-10 [i.8] specifies documentation tags for TTCN-3.

The core language serves three purposes:

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

b) as a standardized interchange format of TTCN-3 test suites between TTCN-3 tools;

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

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

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

TTCN-3 may optionally be used with TTCN-3 packages, which define additional concepts for specific purposes.

TTCN-3 may optionally be used with other type-value notations in which case definitions in other languages may be used as an alternative data type and value syntax. Other parts of the TTCN-3 standard specify use of some other languages with TTCN-3. The support of other languages is not limited to those specified in the ES 201 873 series of documents but to support languages for which combined use with TTCN-3 is defined, rules given in the present document shall apply.

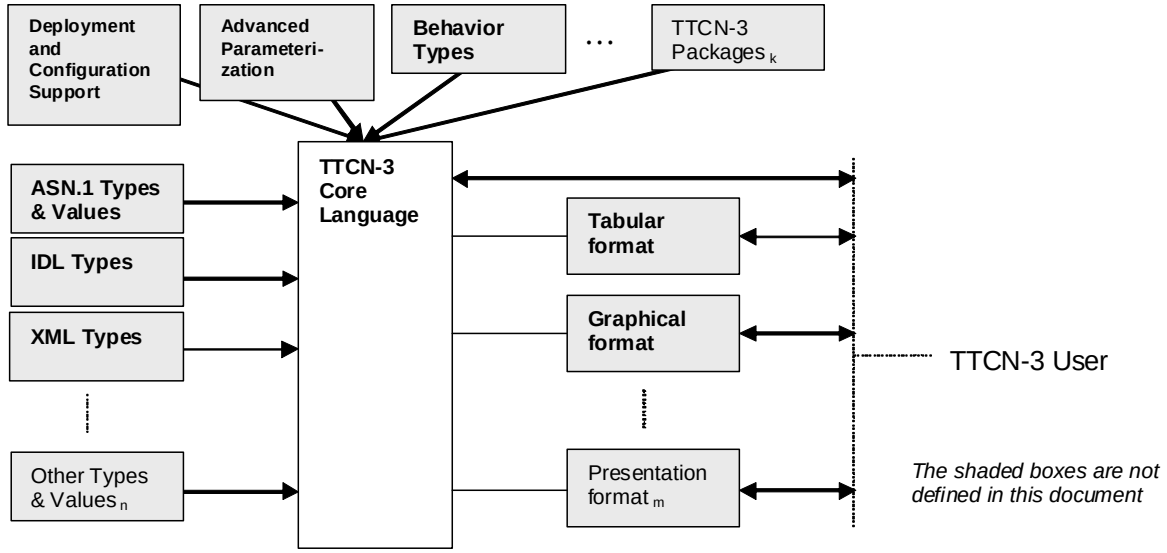


Figure 1: User's view of the core language, its packages and the various presentation formats

The core language is defined by a complete syntax (see annex A) and operational semantics (ES 201 873-4 [1]). It contains minimal static semantics (provided in the body of the present document and in annex A) which do not restrict the use of the language due to some underlying application domain or methodology.

4.2 Unanimity of the specification

The language is specified syntactically and semantically in terms of a textual description in the body of the present document (clauses 5 to 27) and in a formalized way in annex A. In each case, when the textual description is not exhaustive, the formal description completes it. If the textual and the formal specifications are contradictory, the latter shall take precedence.

4.3 Conformance

For an implementation claiming to conform to this version of the language, all features specified in the present document shall be implemented consistently with the requirements given in the present document and in ES 201 873-4 [1].

5 Basic language elements

The top-level unit of TTCN-3 is a module. A module cannot be structured into sub-modules. A module can import definitions from other modules. Modules can have module parameters to allow test suite parameterization.

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

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

TTCN-3 has a number of pre-defined basic data types as well as structured types such as records, sets, unions, enumerated types and arrays.

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

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

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

The TTCN-3 language elements are summarized in table 1.

Table 1: Overview of TTCN-3 language elements

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

5.1 Identifiers and keywords

TTCN-3 identifiers are case sensitive. TTCN-3 keywords shall be written in all lowercase letters (see annex A). TTCN-3 keywords shall neither be used as identifiers of TTCN-3 objects nor as identifiers of objects imported from modules of other languages. The same rules apply to names of predefined TTCN-3 functions (see annex C).

5.2 Scope rules

TTCN-3 provides nine basic units of scope:

a) module definitions part;

b) control part of a module;

c) component types;

d) functions;

e) altsteps;

f) test cases;

g) statement blocks;

h) templates;

i) user defined named types.

NOTE 1: Additional scoping rule for groups is given in clause 8.2.2.

NOTE 2: Additional scoping rule for counters of **`for`** loops is given in clause 19.4.

NOTE 3: Statement blocks may include declarations. They may occur as stand-alone statement blocks, embedded in another statement block or within compound statements, e.g. as body of a **while** loop.

NOTE 4: Built in TTCN-3 types like **`integer`**, **`charstring`**, **`anytype`**, etc. are not scope units, but all named user defined types are scope units, independent of their kinds.

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

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

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

Definitions made in a test component type may be used in a component type extending this component type definition, and in functions, test cases and altsteps referencing that component type or a compatible test component type (see clause 6.3.3) by a **`runs on`**-clause.

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

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

Statement blocks and embedded statement blocks have a hierarchical relation both to the scope unit including the given statement block and to any embedded statement block. Declarations made within a statement block have local visibility.

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

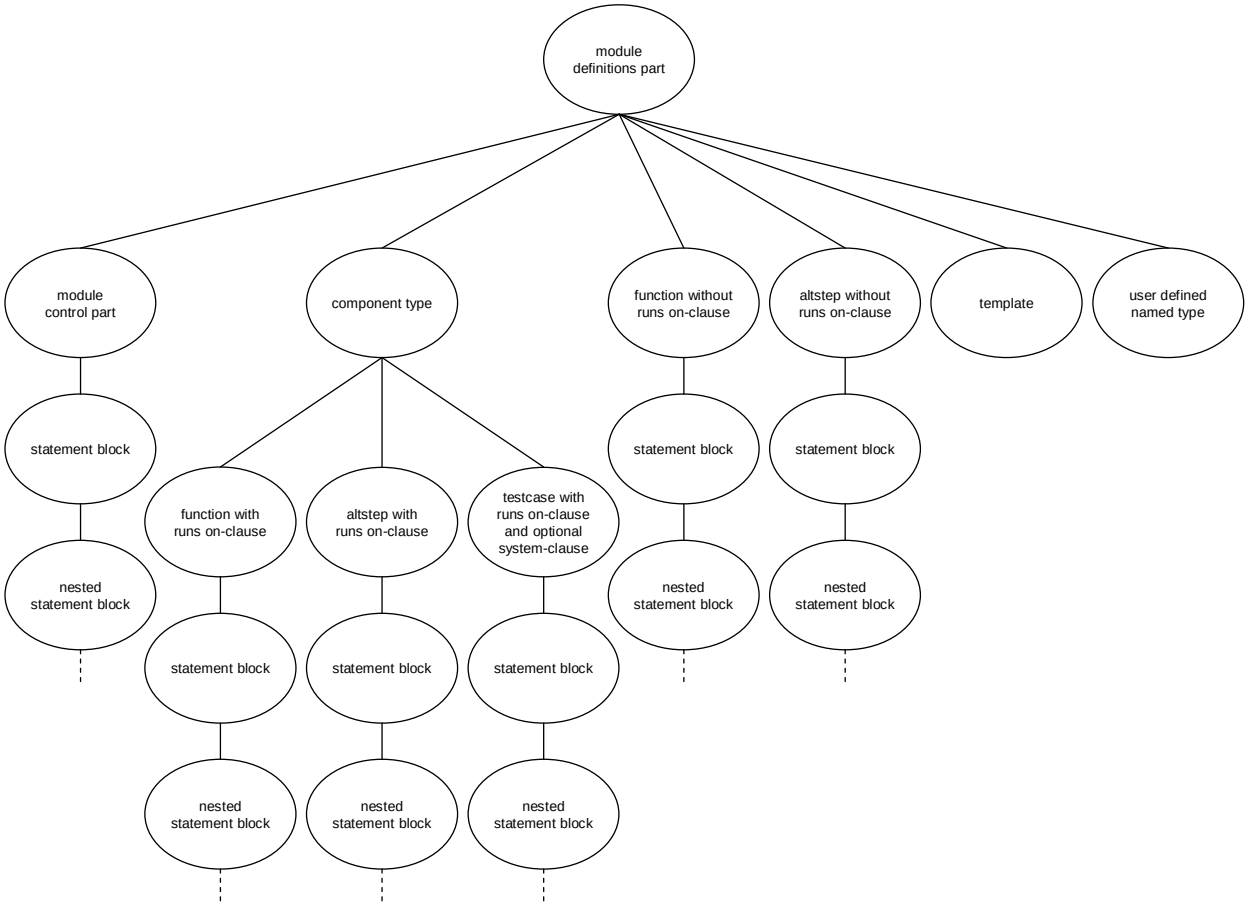


Figure 2: Hierarchy of scope units

EXAMPLE 1: Local scopes

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

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

    }
}
```

EXAMPLE 2: Component type scopes

```
type component MyComponentType {
    const integer MyConst := 1;
    ...
}

type component MyExtendedComponentType extends MyComponentType {
    var integer MyVar:= 2 * MyConst; // using MyConst of MyComponentType
    ...
}
```

5.2.1 Scope of formal parameters

The scope of formal parameters in a parameterized object (e.g. in a function definition) shall be restricted to the definition in which the parameters appear and to the lower levels of scope in the same scope hierarchy. That is they follow the scope rules for local definitions (see clause 5.2).

5.2.2 Uniqueness of identifiers

TTCN-3 requires uniqueness of identifiers, i.e. all identifiers in the same scope hierarchy shall be distinctive. This means that a declaration in a lower level of scope shall not re-use the same identifier as a declaration in a higher level of scope in the same branch of the scope hierarchy.

The identifier of a module (its module name) or of an imported module belongs to the scope unit of the module and cannot be used as identifier for other definitions inside this module. Identifiers for fields of structured types, enumeration values and groups do not have to be globally unique, however in the case of enumeration values the identifiers shall only be reused for enumeration values within other enumerated types. The rules of identifier uniqueness shall also apply to identifiers of formal parameters.

EXAMPLE 1: Nested scopes

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

EXAMPLE 2: Independent scopes

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

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

EXAMPLE 3: Module scopes

```
module MyModuleB {
    import from MyModuleA { … }

    function MyFunction() {
        var integer MyModuleB:= 1; // Is NOT allowed: class with module name
        :
    }

    type boolean MyModuleA; // Is NOT allowed: class with imported module name
}
```

5.3 Ordering of language elements

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

EXAMPLE:

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

Declarations in the module definitions part may be made in any order. However inside the module control part, test case definitions, functions, altsteps, and statement blocks, all required declarations must be given beforehand. This means in particular, local variables, local timers, and local constants shall never be used before they are declared. The only exceptions to this rule are labels. Forward references to a label may be used in **goto** statements before the label occurs (see clause 19.8).

5.4 Parameterization

TTCN-3 allows to parameterize modules, templates, functions, altsteps and testcases. Values, templates, timers, and ports may be used as actual parameters. A summary of which language elements can be parameterized and what can be passed to them as parameters is given in table 2.

NOTE: Type parameterization for TTCN-3 is defined in the optional package [i.12].

Table 2: Overview of parameterizable TTCN-3 objects

Keyword	Allowed kind of Parameterization	Allowed form of Parameterization	Allowed types in formal parameter lists
module	Value parameterization	Static at start of run-time	all basic types, all user-defined types and **address** type.
template	Value and template parameterization	Dynamic at run-time	all basic types, all user-defined types, **address** type and **template.**
function	Value, template, port and timer parameterization	Dynamic at run-time	all basic types, all user-defined types, **address** type, **component** type, **port** type, **default**, **template** and **timer**.
altstep	Value, template, port and timer parameterization	Dynamic at run-time	all basic types, all user-defined types, **address** type, **component** type, **port** type, **default**, **template** and **timer**.
testcase	Value, template, port and timer parameterization	Dynamic at run-time	all basic types and of all user-defined types, **address** type and **template**.
NOTE: Signatures are not shown in the table, because a signature declares parameters only. The templates for the signatures can be parameterized, however.			

5.4.1 Formal parameters

TTCN-3 modules, structured types, templates, functions, altsteps, and testcases may be defined incompletely, i.e. some entities (variables, templates, ports, timers, etc.) used by the above objects may not be resolved in the definition of the object. These objects are called parameterized objects. Formal entities replacing the unresolved entities in the parameterized object's definition are called formal parameters.

Formal parameters of parameterized templates, functions, altsteps, and testcases are defined in formal parameter lists. Formal parameters of modules are defined in module parameter definitions (see clause 8.2.1).

Formal parameters shall be **in**, **inout** or **out** parameters (see definitions in clause 3.1). If not stated otherwise, a formal parameter is an **in** parameter. For all these three sorts of parameter passing, the formal parameters can both be read and set (i.e. get new values being assigned) within the parameterized object. Formal parameters can be used directly as actual parameters for other parameterized objects, e.g. as actual parameters in function invocations or as actual parameters in template instances.

Formal **in** parameters may have default values. This default value is used when no actual parameter is provided.

NOTE: Although **out** parameters can be read within the parameterized object, they do not inherit the value of their actual parameter; i.e. they should be set before they are read.

5.4.1.1 Formal parameters of kind value

Values of all basic types, all user-defined types, address type, component type, and default can be passed as value parameters.

Syntactical Structure

```
[ ( in | inout | out ) ] Type ValueParIdentifier [ ":=" ( Expression | "-" ) ]
```

Semantic Description

Value formal parameters can be used within the parameterized object the same way as values, for example in expressions.

Value formal parameters may be in, inout or out parameters. The default for value formal parameters is **in** parameterization which may optionally be denoted by the keyword **in**. Using of inout or out kind of parameterization shall be specified by the keywords **inout** or **out** respectively.

In parameters may have a default value, which is given by an expression assigned to the parameter. Formal parameters of modified templates may inherit the default values from the corresponding parameters of their parent templates; this shall explicitly be denoted by using a dash (don't change) symbol at the place of the modified template parameters' default value.

TTCN-3 supports value parameterization according to the following rules:

- the language element **module** allows *static* value parameterization to support test suite parameters, i.e. this parameterization may or may not be resolvable at compile-time but shall be resolved by the commencement of run-time (i.e. *static* at run-time). This means that, at run-time, module parameter values are globally visible but not changeable (see more details in clause 8.2);
- the language elements **template**, **testcase, altstep** and **function** support *dynamic* value parameterization (i.e. this parameterization shall be resolved at run-time).

NOTE: Component and default references are also handled as value parameters. In the case of component references, the corresponding component type is the type of the formal parameter. In the case of default references the TTCN-3 type **default** is the type of the formal parameter.

Restrictions

a) Language elements which cannot be parameterized are: **const**, **var**, **timer**, **control**, **record of**, **set of**, **enumerated**, **port**, **component** and subtype definitions, **group** and **import**.

b) Formal value parameters of templates, and of altsteps activated as defaults (see clause 20.5.2) shall always be **in** parameters.

c) Restrictions on module parameters are given in clause 8.2.

d) Default values can be provided for **in** parameters only.

e) The expression of the default value has to be compatible with the type of the parameter. The expression shall not refer to elements of the component type of the optional **runs on** clause. The expression shall not refer to other parameters of the same parameter list. The expression shall not contain the invocation of functions with a **runs on** clause.

f) Default values of component type formal parameters shall be one of the special values **null, mtc, self**, or **system**.

g) Default values of default type formal parameters shall be the special value **null**.

h) The dash (don't change) symbol shall be used with formal parameters of modified templates only (see also clause 15.5).

i) For formal value parameters of templates the restrictions specified in clause 15 shall apply.

Examples

EXAMPLE 1: In, out and inout formal parameters

```
function MyFunction1(in boolean MyReferenceParameter){ … };
// MyReferenceParameter is an in value parameter. The parameter can be read. It can also be set
// within the function, however, the assignment is local to the function only

function MyFunction2(inout boolean MyReferenceParameter){ … };
// MyReferenceParameter is an inout value parameter. The parameter can be read and set
// within the function - the assignment is not local

function MyFunction3(out template boolean MyReferenceParameter){ … };
// MyReferenceParameter is an out value parameter. The parameter can be set within the function,
// the assignment is not local. It can also be read, but only after it has been set.
```

EXAMPLE 2: Reading and setting parameters

```
type record MyMessage {
    integer f1,
    integer f2
}

function f_MyMessage (integer p_int) return MyMessage {
    var integer f1, f2;
    f1 := f_mult2 (p_int);
        // parameter p_int is passed on; as the parameter of the called function f_mult2 is
        // defined as an inout parameter, it passes back the changed value for p_int,
    f2 := p_int;
    return {f1, f2};

}

function f_mult2 (inout integer p_integer) return integer {
    p_integer := 2 * p_integer;
        // the value of the formal parameter is changed; this new value is passed back when
        // f_mult2 completes
    return p_integer-1
}

testcase tc_01 () runs on MTC_PT {
...
    P1.send (f_MyMessage(5))
        // the value sent is { f1 := 9 , f2 := 10 }
...
}
```

EXAMPLE 3: Function with default value for parameter

```
function f_comp (in integer p_int1, in integer p_int2 := 3) return integer {
  var integer v := p_int1 + p_int2;
  :
  return v;
}

function f () {
  var integer w;
  …
  w := f_comp(1);   // same as calling f_comp(1,3);
  w := f_comp(1,2); // value 2 is taken for parameter p_int2 and not its default value 3
  …
}
```

EXAMPLE 4: Direct passing of formal parameters to functions

```
function f_MyFunc2(in bitstring p_refPar1, inout integer p_refPar2) return integer {
    :
}
function f_MyFunc1(inout bitstring p_refPar1, out integer p_refPar2) return integer {
  :
  return f_MyFunc2(p_refPar1, p_refPar2);
}
// p_refPar1 and p_refPar2 can be passed directly to a function invocation
```

5.4.1.2 Formal parameters of kind template

Template kind parameters are used to pass templates into parameterizable objects.

Syntactical Structure

```
[ in | inout | out ] template [ Restriction ] Type ValueParIdentifier
                                                    [ ":=" ( TemplateInstance | "-" ) ]
```

Semantic Description

Templates parameters can be defined for templates, functions, altsteps, and test cases.

To enable a parameterized object to accept templates or matching symbols as actual parameters, the extra keyword **`template`** shall be added before the type field of the corresponding formal parameter. This makes the parameter a template parameter and in effect extends the allowed actual parameters for the associated type to include the appropriate set of matching attributes (see annex B) as well as the normal set of values.

Formal template parameters can be used within the parameterized object the same way as templates and template variables.

Formal template parameters may be in, inout or out parameters. The default for formal template parameters is **`in`** parameterization.

In parameters may have a default template, which is given by a template instance assigned to the parameter. Formal template parameters of modified templates may inherit their default templates from the corresponding parameters of their parent templates; this shall explicitly be denoted by using a dash (don't change) symbol at the place of the modified template parameter's default template.

Formal template parameters can be restricted to accept actual parameters containing a restricted set of matching mechanisms only. Such limitations can be expressed by the restrictions **omit, present**, and **value**. The restriction **template (omit)** can be replaced by the shorthand notation **omit**. The meaning of the restrictions is explained in clause 15.8.

Restrictions

a) Only **`function`, `testcase`**, **`altstep`** and **`template`** definitions may have formal template parameters.

b) Formal template parameters of **`templates`**, of **`function`**s started as test component behaviour (see clause 21.3.2) and of **`altstep`**s activated as defaults (see clause 20.5.2) shall always be **`in`** parameters.

c) Default templates can be provided for in parameters only.

d) The default template instance has to be compatible with the type of the parameter. The template instance shall not refer to elements of the component type in a runs on clause. The template instance shall not refer to other parameters in the same parameter list. The template instance shall not contain the invocation of functions with a runs on clause.

e) Default templates of component type formal parameters shall be built from the special values **`null`, `mtc`, `self`**, or **`system`**.

f) Restrictions specified in clause 15 shall apply.

g) The dash (don't change) symbol shall be used with formal parameters of modified templates only (see also clause 15.5).

Examples

EXAMPLE 1: Template with template parameter

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

// could be used as follows
pco1.receive(MyTemplate(?));
// Or as follows
pco1.receive(MyTemplate(omit)); // provided that field1 is declared in MyMessageType as optional
```

EXAMPLE 2: Function with template parameter

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

EXAMPLE 3: Template with restricted parameter

```
// The template
template MyMessageType MyTemplate1 (template ( omit ) integer MyFormalParam):=
{   field1 := MyFormalParam,
    field2 := pattern "abc*xyz",
    field3 := true
}

// could be used as follows
pco1.send(MyTemplate1(omit));
// but not as follows
pco1.receive(MyTemplate1(?)); // AnyValue is not within the restriction

// the same template can be written shorter as
template MyMessageType MyTemplate2 (omit integer MyFormalParam):=
{   field1 := MyFormalParam,
    field2 := pattern "abc*xyz",
    field3 := true
}
```

5.4.1.3 Formal parameters of kind timer

Functions and altsteps can be parameterized with timers.

Syntactical Structure

```
[ inout ] timer TimerParIdentifier
```

Semantic Description

Timers passed into a parameterized object are known inside the behaviour definition of that object. Timer parameters can be used within the parameterized object like any other timer, i.e. they need not to be declared inside the parameterized object.

Timer parameters shall preserve their current state, i.e. only the timer is made known within the parameterized object. For example, also a started timer continues to run, i.e. it is not stopped implicitly. Thereby, possible timeout events can be handled inside the function or altstep to which the timer is passed.

Formal timer parameters are identified by the keyword **`timer`**.

Restrictions

a) Formal timer parameters shall be inout parameters, which can optionally be indicated by the keyword **inout**.

b) Only **`function`** - with the exception of functions started as test component behaviour (see clause 21.3.2) - and **`altstep`** definitions may have formal timer parameters.

Examples

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

// could be used as follows
function MyBehaviour2 ()
{   :
    timer t;
    MyBehaviour(t);
    :
}
```

5.4.1.4 Formal parameters of kind port

Functions and altsteps can be parameterized with ports.

Syntactical Structure

```
[ inout ] PortTypeIdentifier PortParIdentifier
```

Semantic Description

Ports passed into a parameterized object are known inside the behaviour definition of that object. Port parameters can be used within the parameterized object like any other port, i.e. they need not to be made visible by a **runs on** clause.

Ports passed in as parameters shall preserve their current state, only the port is made known within the parameterized object's body. For example, a started port continues to send/receive messages, i.e. it is not stopped implicitly; thereby, possible port events can be handled inside the function or altstep to which the port is passed to.

Restrictions

a) Formal port parameters shall be inout parameters, which can optionally be indicated by the keyword **inout**.

b) Only **function** - with the exception of functions started as test component behaviour (see clause 21.3.2) - and **altstep** definitions may have formal port parameters.

Examples

```
// Altstep definition with a port in the formal parameter list
altstep MyBehaviour (MyPortType MyPort)
{   :
    [] MyPort.receive { setverdict(fail); stop; }
    :
}
```

5.4.2 Actual parameters

Values, templates, timers and/or ports can be passed into parameterized TTCN-3 objects as actual parameters. Actual parameters can be provided both as a list in the same order as the formal parameters as well as in an assignment notation explicitly using the associated formal parameter names.

Syntactical Structure

```
( Expression |                                                  // for value parameter
  TemplateInstance |                                            // for template parameter
  TimerRef |                                                    // for timer parameter
  Port   |                                                     // for port parameter
  "-" ) |                                                       // to skip a parameter with default
  ParameterId ":=" ( Expression | TemplateInstance | TimerRef | Port ) )
```

Semantic Description

Actual parameters that are passed by value to **in** formal value parameters shall be variables, literal values, module parameters, constants, variables, value returning (external) functions, formal value parameters (of in, inout or out parameterization) of the current scope or expressions composed of the above.

Actual parameters that are passed to **inout** or **out** formal value parameters shall be variables or formal value parameters (of in, inout or out parameterization).

Actual parameters that are passed to **in** formal template parameters shall be literal values, module parameters, constants, variables, value or template returning (external) functions, formal value parameters (of in, inout or out parameterization) of the current scope or expressions composed of the above, as well as templates, template variables or formal template parameters (of in, inout or out parameterization) of the current scope.

Actual parameters that are passed to **inout** or **out** formal template parameters shall be variables, template variables, formal value or template parameters (of in, inout or out parameterization) of the current scope.

Actual parameters that are passed to formal timer parameters shall be component timers, local timers or formal timer parameters of the current scope.

Actual parameters that are passed to formal port parameters shall be component ports or formal port parameters of the current scope.

When a formal parameter has been defined with a default value or template, respectively, then it is not necessary to provide an actual parameter. The actual parameters are evaluated in the order of their appearance. If for some formal parameters, no actual parameter has been provided, their default values are taken and evaluated in the order of the formal parameter list.

The empty brackets for instances of parameterized templates that have only parameters with default values are optional when no actual parameters are provided, i.e. all formal parameters use their default values.

Restrictions

a) When using list notation, the order of elements in the actual parameter list shall be the same as their order in the corresponding formal parameter list. For each formal parameter without a default there shall be an actual parameter. The actual parameter of a formal parameter with default value can be skipped by using dash "-" as actual parameter. An actual parameter can also be skipped by just leaving it out if no other actual parameter follows in the actual parameter list – either because the parameter is last or because all following formal parameters have default values and are left out.

b) Either list notation or assignment notation shall be used in a single parameter list. They shall not be mixed.

c) When using assignment notation, each formal parameter shall be assigned an actual parameter at most once. For each formal parameter without default value, there shall be an actual parameter. In order to use the default value of a formal parameter, no assignment for this specific parameter shall be provided.

d) The type of each actual parameter shall be compatible with the type of each corresponding formal parameter.

e) Actual parameters passed to restricted formal template parameters shall obey the restrictions given in clause 15.8.

f) All parameterized entities specified as an actual parameter shall have their own parameters resolved in the top-level actual parameter list.

g) If the formal parameter list of TTCN-3 objects **function**, **testcase**, **signature, altstep** or **external function** is empty, then the empty parentheses shall be included both in the declaration and in the invocation of that object. In all other cases the empty parentheses shall be omitted.

h) Restrictions on the use of signature parameters are given in clauses 15.2 and 22.3.

i) Restrictions on parameters passed to altsteps are given in clauses 16.2.1 and 20.5.2.

Examples

EXAMPLE 1: Formal and actual parameter lists have to match

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

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

// A function call with assignment notation for actual parameters
MyFunction(FormalPar1 := 123, FormalPar3 := '1100'B, FormalPar2 := true);
```

EXAMPLE 2: In parameters

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

// A function call with an actual parameter
MyFunction(MyGlobalTemplate);
```

EXAMPLE 3: Inout and out parameters

```
function MyFunction(inout boolean MyReferenceParameter){ … };
// MyReferenceParameter is an inout parameter

// A function call with an actual parameter
MyFunction(MyBooleanVariable);
// The actual parameter can be read and set within the function


function MyFunction(out template boolean MyReferenceParameter){ … };
// MyReferenceParameter is an out parameter

// A function call with an actual parameter
MyFunction(MyBooleanVariable);
// The actual parameter is initially unbound, but can be set and read within the function.
```

EXAMPLE 4: Empty parameter lists

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

// and shall be called as
MyFunction();


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

// and shall be used as
template MyRecord Mytemplate := { … }
```

EXAMPLE 5: Nested parameter lists

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

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

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

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

5.5 Cyclic Definitions

Direct and indirect cyclic definitions are not allowed with the exception of the following cases:

a) for recursive type definitions (see clause 6.2);

b) function and altstep definitions (i.e. recursive function or altstep calls);

c) cyclic import definitions, if the imported definitions only form allowed cyclic definitions.

NOTE 1: Indirect cyclic definitions may be a result of imports of definitions that are needed for the usage of a definition but do not need to be known in the importing module (see clause 8.2.3.1).

NOTE 2: For the detection of cycles only the main identifiers of the definition are used. For example, field identifiers are not used.

Examples

EXAMPLE 1: Module with cyclic constant definition that is not allowed

```
module MyModule {
    :
    type record ARecordType { integer a, integer b };
    :
// The following two lines include a cycle that is not allowed
    const ARecordType cConst := { 1 , dConst.b}; // cConst refers to dConst
    const ARecordType dConst := { 1 , cConst.b}; // dConst refers to cConst
}
```

EXAMPLE 2: Modules with cyclic import that is allowed

```
module MyModuleA {
        import from MyModuleB { type MyInteger }
        type record of MyInteger MyIntegerList;
}

module MyModuleB {
        type integer MyInteger;
        import from MyModuleA { type MyIntegerList }
}
```

6 Types and values

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

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

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

The special type **default** may be used for the default handling (see clause 20.5).

The TTCN-3 types are summarized in table 3.

Table 3: Overview of TTCN-3 types

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

NOTE: Behaviour types for TTCN-3 are defined in the optional package [i.13].

6.1 Basic types and values

6.1.0 Simple basic types and values

TTCN-3 supports the following basic types:

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

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

b) **`float`:** a type to describe floating-point numbers and special float values.

In general, floating point numbers can be defined as:<*mantissa*> × <*base*>$^{<exponent>}$,

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

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

- the decimal notation with a dot in a sequence of numbers like, 1.23 (which represents 123×10^{-2}), 2.783 (i.e. 2783×10^{-3}) or -123.456789 (which represents $-123\ 456\ 789 \times 10^{-6}$); or
- by two numbers separated by E where the first number specifies the mantissa and the second specifies the exponent, for example 12.3E4 (which represents 123×10^{3}) or -12.3E-4 (which represents -123×10^{-5}).

NOTE 1: In contrast to the general definition of float values, the mantissa of in theTTCN-3 value notation, beside integers, allows decimal numbers as well.

The special values of the float type consist of **infinity** (positive infinity), **-infinity** (negative infinity) and the value **not_a_number**. For the ordering of special values see clauses 7.1.1 and 7.1.3.

NOTE 2: **- not_a_number** (i.e. minus not a number) is not to be used.

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

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

d) **verdicttype:** a type for use with test verdicts consisting of 5 distinguished values. Values of **verdicttype** shall be denoted by **pass**, **fail**, **inconc**, **none** and **error**.

6.1.1 Basic string types and values

TTCN-3 supports the following basic string types:

NOTE 1: The general term string or string type in TTCN-3 refers to **bitstring**, **hexstring**, **octetstring**, **charstring** and **universal charstring**.

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

Values of type **bitstring** shall be denoted by an arbitrary number (possibly zero) of the bit digits: 0 1, preceded by a single quote (') and followed by the pair of characters 'B.

EXAMPLE 1: `'01101'B`.

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

Values of type **hexstring** shall be denoted by an arbitrary number (possibly zero) of the hexadecimal digits (uppercase and lowercase letters can equally be used as hex digits):

0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

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

EXAMPLE 2:
```
'AB01D'H
'ab01d'H
'Ab01D'H
```

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

Values of type **octetstring** shall be denoted by an arbitrary, but even, number (possibly zero) of the hexadecimal digits (uppercase and lowercase letters can equally be used as hex digits):

0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

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

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

d) **charstring:** are types whose distinguished values are zero, one, or more characters of the version of ITU-T Recommendation T.50 [4] complying with the International Reference Version (IRV) as specified in clause 8.2 of ITU-T Recommendation T.50 [4].

NOTE 2: The IRV version of ITU-T Recommendation T.50 [4] is equivalent to the IRV version of the International Reference Alphabet (former International Alphabet No.5 - IA5), described in ITU-T Recommendation T.50 [4].

Values of **charstring** type shall be denoted by an arbitrary number (possibly zero) of non-control characters from the relevant character set, preceded and followed by double quote ("). Graphical characters include the range from SP(32) to TILDE (126). Values of **charstring** type can also be calculated using the predefined conversion function int2char with the positive integer value of their encoding as argument (see clause C.1).

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

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

EXAMPLE 4: The charstring "ab"cd" is written in TTCN-3 code as in the following constant declaration. Each of the 3 quote characters that are part of the string is preceded by an extra quote character and the whole character string is delimited by quote characters, e.g.

```
var charstring vl_char:= """ab""cd""";
```

e) The character string type preceded by the keyword **universal** denotes types whose distinguished values are zero, one, or more characters from ISO/IEC 10646 [2].

universal charstring values can also be denoted by an arbitrary number (possibly zero) of characters from the relevant character set, preceded and followed by double quote ("), calculated using a predefined conversion function (see clause C.2) with the positive integer value of their encoding as argument or by a "quadruple".

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

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

The "quadruple" is only capable to denote a single character and denotes the character by the decimal values of its group, plane, row and cell according to ISO/IEC 10646 [2], preceded by the keyword **char** included into a pair of brackets and separated by commas (e.g. **char** (0, 0, 1, 113) denotes the Hungarian character "ű"). In cases where it is necessary to denote the character double quote (") in a string assigned according to the first method (within double quotes), the character is represented by a pair of double quotes on the same line with no intervening space characters. The two methods may be mixed within a single notation for a string value by using the concatenation operator.

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

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

By default, **universal charstring** shall conform to the UCS-4 coded representation form specified in clause 14.2 of ISO/IEC 10646 [2].

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

This default encoding can be overridden using the defined variant attributes (see clause 27.5). The following useful character string types utf8string, bmpstring, utf16string and iso8859string using these attributes are defined in annex E.

6.1.1.1 Accessing individual string elements

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

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

Indexing shall begin with the value zero (0). The index shall be between zero and the length of the string minus one for retrieving an element from a string. For assigning an element to the end of a string, the length of the string should be used as index.

EXAMPLE 1: Accessing an existing element

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

EXAMPLE 2: Specific cases

```
var bitstring MyBitStringA, MyBitStringB, MyBitStringC;
MyBitStringA := '010'B;
MyBitStringA[1] := '11'B; //causes an error as only individual elements can be accessed

MyBitStringB := '1'B;
MyBitStringB[4] := '1'B; //causes an error as the index is larger than the length of the lhs

MyBitStringC := ''B;
MyBitStringC[0] := '1'B; // value of MyBitStringC is '1'B
MyBitStringC[1] := '0'B; // value of MyBitStringC is '10'B
```

6.1.2 Subtyping of basic types

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

6.1.2.1 Lists of values

TTCN-3 permits the specification of a list of distinguished values as listed in table 3. The values in the list shall be instances of the type being constrained and shall be a subset of the values defined by the type being constrained. The subtype defined by this list restricts the allowed values of the subtype to those values in the list. Constants used in the constant expressions defining the values shall meet with the restrictions in clause 10.

EXAMPLE:

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

6.1.2.2 Lists of types

TTCN-3 permits the specification of a list of subtypes as listed in table 3 for value lists. The types in the list shall be subtypes of the root type. The subtype defined by this list restricts the allowed values of the subtype to the union of the values of the referenced subtypes.

EXAMPLE:

```
type bitstring BitStrings1 ('0'B, '1'B );
type bitstring BitStrings2 ('00'B, '01'B, '10'B, '10'B);
type bitstring BitStrings_1_2 (Bitstrings1, Bitstrings2);
```

6.1.2.3 Ranges

TTCN-3 permits the specification of range constraints for the types **integer, charstring, universal charstring** and **float** (or derivations of these types). For **integer** and **float**, the subtype defined by the range restricts the allowed values of the subtype to the values in the range including or excluding the lower boundary and/or the upper boundary. The upper boundary shall be greater than or equal to the lower boundary.

In order to specify an infinite integer range, the keyword **-infinity** or **infinity** can be used instead of a value indicating that there is no lower or upper boundary; **-infinity** shall not be used as the upper bound and **infinity** shall not be used as the lower bound for integer ranges.

Also for **float**, **-infinity** or **infinity** can be used as the bounds in range restrictions. Using the special value **-infinity** as the lower bound shall indicate that the allowed numerical values are not restricted downward and the special value **-infinity** is also included. If both the lower and upper bounds denote **-infinity**, no numerical values are included, only the special value **-infinity**. Using the special value **infinity** as the upper bound shall indicate that the allowed numerical values are not restricted upward and the special value **infinity** is also included. If both the lower and upper bounds denote **infinity**, no numerical values are included, only the special value **infinity**. If exclusive bounds (!**infinity** or !**-infinity**) is used instead, only the respective numerical float values are included in the range. In case of **float**, the special value **not_a_number** is not allowed in a range constraint.

In the case of **charstring** and **universal charstring** types, the range restricts the allowed values for each separate character in the strings. The boundaries shall evaluate to valid character positions according to the coded character set table(s) of the type (e.g. the given position shall not be empty). Empty positions between the lower and the upper boundaries are not considered to be valid values of the specified range.

Constants used in the constant expressions defining the values shall meet with the restrictions in clause 10.

EXAMPLE 1:

```
type integer MyIntegerRange (0 .. 255); );      // range from 0..255
                                                // (with inclusive boundaries)
type integer MyIntegerRange (-infinity .. -1); // all negative integer numbers
type integer MyIntegerRange (0 .. !256);        // the same range as above (with left
                                                // inclusive and right exclusive boundary)
type integer MyIntegerRange (!-1 .. 255);       // the same range as above(with left
                                                // exclusive and right inclusive boundary)
type integer MyIntegerRange (!-1 .. !256);      // the same range as above
                                                // (with exclusive boundaries)
type float piRange (3.14 .. 3142E-3);
type float LessThanPi (-infinity .. 3142E-3);
type float Numbers (-infinity .. infinity);     //includes all float values but not_a_number
type float Wrong (-infinity .. not_a_number);   // causes an error as not_a_number is not
                                                // allowed in range subtyping
```

EXAMPLE 2:

```
type charstring MyCharString ("a" .. "z");
// Defines a string type of any length with each character within the specified range
type universal charstring MyUCharString1 ("a" .. !"z");
// Defines a string type of any length with each character within the range from a to y
// (character codes from 97 to 121), like "abxy";
// strings containing any other character (including control characters), like
// "abc2" are disallowed.
type universal charstring MyUCharString2 (char(0, 0, 1, 111) .. char(0, 0, 1, 113));
// Defines a string type of any length with each character within the range specified using
// the quadruple notation
```

6.1.2.4 String length restrictions

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

EXAMPLE:

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

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

Table 4: Units of length used in field length specifications

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

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

6.1.2.5 Pattern subtyping of character string types

TTCN-3 allows using character patterns specified in clause B.1.5 to constrain permitted values of **charstring** and **universal charstring** types. The type constraint shall use the **pattern** keyword followed by a character pattern. All values denoted by the pattern shall be a subset of the type being subtyped. Constants used in the constant expressions defining the values shall meet with the restrictions in clause 10.

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

EXAMPLE:

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

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

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

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

6.1.2.6 Mixing subtyping mechanisms

6.1.2.6.1 Mixing patterns, lists and ranges

Within **integer** and **float** (or derivations of these types) subtype definitions it is allowed to mix lists and ranges. It is possible to mix both value list and type list subtyping with each other and with range subtyping. Overlapping of different constraints is not an error.

EXAMPLE 1:

```
type integer MyIntegerRange (1, 2, 3, 10 .. !20, 99, 100);
type float lessThanPiAndNaN (-infinity .. 3142E-3, not_a_number);
```

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

EXAMPLE 2:

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

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

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

6.1.2.6.2 Using length restriction with other constraints

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

Within **charstring** and **universal charstring** subtype definitions it is allowed to add a length restriction to constraints containing list, range or pattern subtyping in the same subtype definition.

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

EXAMPLE:

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

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

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

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

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

type charstring MyCharStr10 ("gr", "xyz") length (4);
    // causes an error as it contains no value
```

6.2 Structured types and values

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

Values of these types may be given using an explicit assignment notation or a short-hand value list notation.

EXAMPLE 1:

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

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

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

EXAMPLE 2:

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

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

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

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

EXAMPLE 3:

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

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

EXAMPLE 4:

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

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

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

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

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

In case of record and set types, to avoid infinite recursion, fields referencing to its own type, shall be optional.

EXAMPLE 5:

```
// Valid recursive record type definition
type record MyRecord1
{
    FieldType1  field1,
    MyRecord1   field2 optional,
    FieldType3  field3
}

// Invalid recursive record type definition causing an error
type record MyRecord2
{
    FieldType1  field1,
    MyRecord2   field2,
    FieldType3  field3
}
```

In case of union types, to avoid infinite recursion, at least one of the alternatives shall not reference its own type.

EXAMPLE 6:

```
// Valid recursive union type definition
type union MyUnion1
{
    MyUnion1   choice1,
    charstring choice2
}

// Invalid recursive union type definition causing an error
type union MyUnion2
{
    MyUnion2 choice1,
    MyUnion2 choice2
}
```

6.2.1 Record type and values

TTCN-3 supports ordered structured types known as **record**. The elements of a **record** type may be any of the basic types or user-defined data types (such as other records, sets or arrays). The values of a **record** shall be compatible with the types of the **record** fields. The element identifiers are local to the **record** and shall be unique within the **record** (but do not have to be globally unique).

EXAMPLE 1:

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

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

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

EXAMPLE 2:

```
type record MyEmptyRecord {}
```

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

EXAMPLE 3:

```
var integer MyIntegerValue := 1;

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

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

The same value specified with a value list.

EXAMPLE 4:

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

6.2.1.1 Referencing fields of a record type

Elements of a **record** shall be referenced by the dot notation *TypeIdOrExpression.ElementId*, where *TypeIdOrExpression* resolves to the name of a structured type or an expression of a structured type such as variable, formal parameter, module parameter, constant, template, or function invocation. *ElementId* shall resolve to the name of a field in the structured type. Fields of record type definitions shall not reference themselves.

EXAMPLE 1:

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

EXAMPLE 2:

```
type record MyType
{
    integer field1,
    MyType.field2   field2  optional, // this circular reference is NOT ALLOWED
    boolean field3
}
```

6.2.1.2 Optional elements in a record

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

EXAMPLE 1:

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

Optional fields shall be omitted using the omit symbol.

EXAMPLE 2:

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

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

6.2.1.3 Nested type definitions for field types

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

EXAMPLE:

```
// record type with nested structured type definitions
type record MyNestedRecordType
{
    record
    {
        integer nestedField1,
        float nestedField2
    } outerField1,
    enumerated {
        nestedEnum1,
        nestedEnum2
    } outerField2,
    record of boolean outerField3
}

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

6.2.2 Set type and values

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

EXAMPLE:

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

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

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

6.2.2.1 Referencing fields of a set type

Elements of a **set** shall be referenced by the dot notation (see clause 6.2.1.1). Elements of set type definitions shall not reference themselves.

EXAMPLE:

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

6.2.2.2 Optional elements in a set

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

6.2.2.3 Nested type definition for field types

TTCN-3 supports the definition of types for set fields nested within the **set** definition, similar to the mechanism for record types described in clause 6.2.1.3.

6.2.3 Records and sets of single types

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

NOTE 1: Subtyping of record of and set of types see in clause 6.2.13.

EXAMPLE 1:

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

The value notation for **record of** and **set of** can be both the value list notation and the assignment notation (usable to address multiple elements) or an indexed notation (usable to address an individual element), which is the same value notation as for arrays (see clause 6.2.7). There is one exception from this general rule: in the case of defining modified templates, the assignment notation is also allowed to be used (see clause 15.5).

When the value list notation is used, the first value in the list is assigned to the first element, the second list value is assigned to the second element, etc. No empty assignment is allowed (e.g. two commas, the second immediately following the first or only with white space between them). Elements to be left out of the assignment shall be explicitly skipped in the list by use of the not-used-symbol "-".

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

EXAMPLE 2:

```
// Given
type record of integer MyRecordOf;
var integer MyVar;
// Using the value list notation
var MyRecordOf MyRecordOfVar := { 0, 1, 2, 3, 4 };

// The same record of, defined with the assignment notation
var MyRecordOf MyRecordOfVarAssignment := {
    [0] := 0,
    [1] := 1,
    [2] := 2,
    [3] := 3,
    [4] := 4
};

//Using an indexed notation
MyVar := MyRecordOfVar[0]; // the first element of the "record of" value (integer 0)
                           // is assigned to MyVar

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

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

// The assignment
MyRecordOfVar[6] := 6;
// will change the value of MyRecordOfVar to{ 0, 1, 2 , 2, <undefined>,   <undefined>, 6 };
// Note the 5th and 6th elements (with indexes 4 and 5) had no assigned value before this
// last assignment and are therefore undefined.

MyRecordOfVar[4] := 4; MyRecordOfVar[5] := 5;
// will complete MyRecordOfVar to the fully defined value { 0, 1, 2 , 2, 4 , 5 , 6 };
```

NOTE 2: The index notation makes it possible e.g. to copy **record of** values element by element in a for loop. For example, the function below reverses the elements of a **record of** value:

```
function reverse(in MyRecordOf src) return MyRecordOf
{
var MyRecordOf dest;
var integer i, srcLength := lengthof (src);
for(i := 0; i < srcLength; i:= i + 1) {
    dest[srcLength - 1 - i] := src[i];
}
return dest;
}
```

Embedded **record of** and **set of** types will result in a data structure similar to multidimensional arrays (see clause 6.2.7).

EXAMPLE 3:

```
// Given
type record of integer MyBasicRecordOfType;
type record of MyBasicRecordOfType  My2DRecordOfType;

// Then, the variable myRecordOfArray will have similar attributes to a two-dimensional array:
var My2DRecordOfType myRecordOfArray;
// and reference to a particular element would look like this
// (value of the second element of the third 'MyBasicRecordOfType' construct)
myRecordOfArray [2][1] := 1;
```

6.2.3.1 Nested type definitions

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

EXAMPLE:

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

6.2.3.2 Referencing elements of record of and set of types

It is also allowed to reference the inner type of **record of** and **set of** types by using the index notation but with a dash. The notation *TypeId[-]*, where *TypeId* resolves to the name of a **record of** or **set of** type, references the inner type of *TypeId*.

EXAMPLE:

```
//Provided the definitions below
type record of integer MyRecordOfInt;
type record of record {
    integer f1,
    set { integer s1, boolean s2 } f2
} MyRecordOfRecord;
type record of record of integer MyRecordOfRecordOfInt;
type record of record {
    integer f1,
    record of boolean f2
} MyRecordOfRecord2;

// Referencing the inner integer type
type MyRecordOfInt[-] MyInteger;
const MyRecordOfInt[-] c_MyInteger:= 5;

// Referencing the nested record type
type MyRecordOfRecord[-] MyInnerRecord;
const MyRecordOfRecord[-] c_MyRecord := { f1 = 5; f2 := { s1 := 0; s2 := true }}

// Referencing the set type nested in the inner record
type MyRecordOfRecord[-].f2 MyNestedSet;
const MyRecordOfRecord[-].f2 c_MySet := { s1 := 0; s2 := true }
```

```
// Referencing the innermost boolean
type MyRecordOfRecord[-].f2.s2 MyBoolean;
const MyRecordOfRecord[-].f2.s2 c_MyBool := false;

// Referencing the inner record of
type MyRecordOfRecordOfInt[-] MyInnerRecordOfInt;
const MyRecordOfRecordOfInt[-] c_MyInnerRecordOfInt := { 0, 1, 2, 3 };

// Referencing the integer type within the inner record of
type MyRecordOfRecordOfInt[-][-] MyInteger2;
const MyRecordOfRecordOfInt[-][-] c_MyInteger2 := 1;

// Referencing the boolean type within the nested record
type MyRecordOfRecord2[-].f2[-] MyInnermostBoolean;
const MyRecordOfRecord2[-].f2[-] c_MyInnermostBoolean := true ;
```

6.2.4 Enumerated type and values

TTCN-3 supports **enumerated** types. Enumerated types are used to model types that take only a distinct named set of values. Such distinct values are called enumerations. Each enumeration shall have an identifier. Operations on enumerated types shall only use these identifiers and are restricted to assignment, equivalence and ordering operators. Enumeration identifiers shall be unique within the enumerated type (but do not have to be globally unique) and are consequently visible in the context of the given type only. Enumeration identifiers shall only be reused within other structured type definitions and shall not be used for identifiers of local or global visibility at the same or a lower level of the same branch of the scope hierarchy (see scope hierarchy in clause 5.2).

NOTE 1: "In the context" means that at least one object involved in the given TTCN-3 action (an assignment, operation, parameter passing etc.) identifies a concrete type unambiguously, i.e. either directly (e.g. an in-line template) or by means of a typed TTCN-3 object (e.g. via a constant, variable, formal parameter etc.).

EXAMPLE 1: Declaration of enumerated types and values

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

type integer Monday;
// This definition does not clash with the previous one
// as Monday in MyFirstEnumType is of local scope

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

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

type record MyNewRecordType {
    MyFirstEnumType firstField,
    integer         secondField
};

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

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

Each enumeration may optionally have a user-assigned integer value, which is defined after the name of the enumeration in parenthesis. Each user-assigned integer number shall be distinct within a single **enumerated** type. For each enumeration without an assigned integer value, the system successively associates an integer number in the textual order of the enumerations, starting at the left-hand side, beginning with zero, by step 1 and skipping any number occupied in any of the enumerations with a manually assigned value. These values are only used by the system to allow the use of relational operators. The user shall not directly use associated integer values but can access them and convert integer values into enumeration values by using the predefined functions **enum2int** and **int2enum** (see clauses 16.1.2 and C.37).

NOTE 2: The integer value also may be used by the system to encode/decode enumerated values. This, however is outside the scope of the present document (with the exception that TTCN-3 allows the association of encoding attributes to TTCN-3 items).

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

NOTE 3: If the enumerated type is an element of a user defined structured type, the enumerated type is implicitly referenced via the given element (i.e. by the identifier of the element or the position of the value in a value list notation) at value assignment, instantiation etc.

EXAMPLE 2: Using enumerated types (see also example 4 of clause 8.2.3)

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

// The following statements however cause an error as the two variables are instances
// of different enumeration types
Today := Tomorrow;
Today == Tomorrow;

// The following operation is correct
if (Today == Monday ) {...}
// the type of variable Today identifies the type context of MyFirstEnumType for the
// equality operator

// But the following causes an error
if ( Tuesday == Wednesday ) {...}
// there is no TTCN-3 type(d) object to establish the type context for the equality operator
// Please note that the values Tuesday and Wednesday are defined within the type
// MyFirstEnumType only, but this is not sufficient to establish the type context
```

When a TTCN-3 global or local definition is declared using an imported enumerated type, the name of that definition shall not be the same as any of the enumeration value names of that type.

6.2.5 Unions

TTCN-3 supports the **union** type. The **union** type is a collection of alternatives, each one identified by an identifier. Only one of the specified alternatives will ever be present in an actual union value. Union types are useful to model data which can take one of a finite number of known types.

EXAMPLE:

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

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

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

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

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

6.2.5.1 Referencing fields of a union type

Alternatives of a **union** type shall be referenced by the dot notation (see clause 6.2.1.1). Alternatives of union type definitions shall not reference themselves.

EXAMPLE:

```
MyVar5 := MyUnion1.myChoice1;
```

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

6.2.5.2 Option and union

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

6.2.5.3 Nested type definition for field types

TTCN-3 supports the definition of types for union alternatives nested within the union definition, similar to the mechanism for record types described in clause 6.2.1.3.

6.2.6 The anytype

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

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

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

EXAMPLE:

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

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

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

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

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

6.2.7 Arrays

Arrays can be used in TTCN-3 as a shorthand notation to specify record of types. They may be specified also at the point of a variable declaration. Arrays may be declared as single or multi-dimensional. Array dimensions shall be specified using constant expressions, which shall evaluate to a positive **integer** values. Constants used in the constant expressions shall meet with the restrictions in clause 10.

EXAMPLE 1:

```
type integer MyArrayType1[3];                      // A type with 3 integer elements
type record length (3) of integer MyRecordOfType1; // The corresponding record of

var MyArrayType1 a1:= { 7, 8, 9 };
var MyRecordOfType1 r1:= a1;    // MyArrayType1 and MyRecordOfType1 are compatible

var integer myArray1[3]:= r1;   // Instantiates an integer array of 3 elements
                                // with the index 0 to 2
                                // being compatible to MyArrayType1 and MyRecordOfType1
```

```
var integer myArray2[2][3]; // Instantiates a two-dimensional integer array of 2 × 3 elements
                            // with indexes from (0,0) to (1,2)
```

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

EXAMPLE 2:

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

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

Array dimensions may also be specified using ranges (with inclusive boundaries only). In such cases, the lower and upper values of the range define the lower and upper index values. Such an array is corresponding to a record of with a fixed length restriction computed as the difference between upper and lower index bound plus 1 and indexing starting from the lower bound of the array definition.

EXAMPLE 3:

```
type integer MyArrayType2[2 .. 5]; // A type with 4 integer elements, indices starting with 2
type record length (4) of integer MyRecordOfType2; // The corresponding record of

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

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

NOTE: It is not possible to define an array type with a variable amount of elements. Neither is it possible to define an unlimited array with a lower bound on the array index.

The values of array elements shall be compatible with the corresponding variable or type declaration. Values may be assigned individually by a value list notation or indexed notation or more than one or all at once by a value list notation. When the value list notation is used, the first value of the list is assigned to the first element of the array (the element with index 0 or the lower bound if an index range has been given), the second value to the next element, etc. Elements to be left out from the assignment shall be explicitly skipped in the list by using dash.

Indexed value notation can be used on both the right-hand side and left-hand side of assignments. The index of the first element shall be zero or the lower bound if an index range has been given. The index shall not exceed the limitations given by either the length or the upper bound of the index. If the value of the element indicated by the index at the right-hand of an assignment is undefined, this shall cause an error. Sending an array value with undefined elements shall cause an error. All elements in an array value that are not set explicitly, are undefined.

For assigning values to multi-dimensional arrays, each dimension that is assigned shall resolve to a set of values enclosed in curly braces. When specifying values for multi-dimensional arrays, the leftmost dimension corresponds to the outermost structure of the value, and the rightmost dimension to the innermost structure. The use of array slices of multi-dimensional arrays, i.e. when the number of indexes of the array value is less than the number of dimensions in the corresponding array definition, is allowed. Indexes of array slices shall correspond to the dimensions of the array definition from left to right (i.e. the first index of the slice corresponds to the first dimension of the definition). Slice indexes shall conform to the related array definition dimensions.

EXAMPLE 4:

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

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

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

var integer MyArray5[2][3][4] :=
{
    {
```

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

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

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

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

6.2.8 The default type

TTCN-3 allows the activation of altsteps (see clause 16.2) as defaults to capture recurring behaviour. Default references are unique references to activated defaults. Such a unique default reference is generated by a test component when an altstep is activated as a default, i.e. a default reference is the result of an **activate** operation (see clause 20.5.2).

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

Default references are used in **deactivate** operations (see clause 20.5.3) in order to identify the default to be deactivated.

Default references have meaning only within the test component instances they are activated, i.e. a default reference assigned to a default variable in test component instance "a1" of type "A" has no meaning in test component instance "a2" of type "A".

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

6.2.9 Communication port types

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

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

Ports are bidirectional. The directions are specified by the keywords **in** (for the in direction), **out** (for the out direction) and **inout** (for both directions). Directions shall be seen from the point of view of the test component owning the port with the exception of the test system interface, where **in** identifies the direction of message sending or procedure call and **out** identifies the direction of message receive, get reply or catch exception from the point of view of the test component connected to the test system interface port.

Each port type definition shall have one or more lists indicating the allowed collection of (message) types or procedure signatures together with the allowed communication direction.

For configuration purposes the port type may have one **map param** and one **unmap param** declaration indicating the allowed additional parameters for the respective operation. These formal parameters must be value parameters.

Whenever a signature (see also clause 14) is defined in the **out** direction of a procedure-based port, the types of all its **inout** and **out** parameters, its return type and its exception types are automatically part of the **in** direction of this port. Whenever a signature is defined in the **in** direction for a procedure-based port, the types of all its **inout** and **out** parameters, its return type and its exception types are automatically part of the **out** direction of this port.

Ports used for the communication with the SUT may need to address specific entities within the SUT. In addition, several address schemes may be supported by one SUT at different ports. To support such addressing schemes, TTCN-3 allows to bind an **address** type to a port. Values of this type may be used for addressing purposes in communication operations (see clause 22.1) and be stored in variables. The handling of address types bound to different ports by means of the dot notation is explained in clause 6.2.12.

Syntactical Structure

Message-based port:

```
type port PortTypeIdentifier message "{"
     [ address Type ";" ]
     [ map param "(" { FormalValuePar [","] }+ ")" ]
     [ unmap param "(" { FormalValuePar [","] }+ ")" ]
     { ( in | out | inout ) { MessageType [ "," ] }+ ";" }
"}"
```

Procedure-based port:

```
type port PortTypeIdentifier procedure "{"
     [ address Type ";" ]
     [ map param "(" { FormalValuePar [","] }+ ")" ]
     [ unmap param "(" { FormalValuePar [","] }+ ")" ]
     { ( in | out | inout ) { Signature [ "," ] }+ ";" }
"}"
```

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

EXAMPLE 1: Message-based port

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

EXAMPLE 2: Procedure-based port

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

EXAMPLE 3: Message-based port with address type definition

```
type port MyMessagePortTypeTwo message
{
    address integer;        // if addressing is used on ports of type MyMessagePortTypeTwo
                        // the addresses have to be of type integer
    inout   MsgType1, MsgType2;
}
```

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

EXAMPLE 4: Usage of param in port declaration

```
// Message based port which allows MsgType4 to be send and received over the port
// and MsgType5 and MsgType6 as configuration parameter type
type port MyMessagePortType message
{
    inout   MsgType4;
    map param   (in MsgType5 p1, out MsgType6 p2);
}

// Procedure based port which allows the remote call of the procedure Proc1
// and MsgType5 as configuration parameter type
type port MyProcedurePortType procedure
{
    out     Proc1;
    unmap param (MsgType5 p1);
}
```

6.2.10 Component types

6.2.10.1 Component type definition

The component type defines which ports are associated with a component (see figure 3). The port names in a component type definition are local to that component type, i.e. another component type may have ports with the same names. Port names in the same component type definition shall all have unique names.

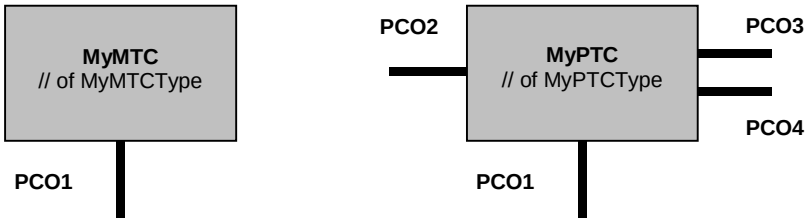


Figure 3: Typical components

It is also possible to declare constants, variables and timers local to a particular component type. These declarations are visible to all testcases, functions and altsteps that run on an instance of the given component type. This shall be explicitly stated using the **runs on** keyword (see clause 16) in the testcase, function or altstep header. Component type definitions are associated with the component instance and follow the scope rules defined in clause 5.2. Each new instance of a component type will thus have its own set of constants, variables and timers as specified in the component type definition (including any initial values, if stated). Constants used in the constant expressions of type declarations for variables, constants or ports shall meet with the restrictions in clause 10, however constants used in the constant expressions of initial values for variables, constants or timers do not have to obey these restrictions.

Syntactical Structure

```
type component ComponentTypeIdentifier "{"
     { ( PortInstance
     | VarInstance
     | TimerInstance
     | ConstDef ) }
"}"
```

Semantic Description

Component type definitions specify the creation, declaration and initialization of ports and component constants, variables and timers during the creation of an instance of a component type. These instances can be used as the main test component, as the test system interface or as a parallel test component. Every instance of a component type has its own fresh copy of the port, constant, variable, and timer instances defined in the component type definition.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

EXAMPLE 1: Component type with port instances only

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

EXAMPLE 2: Component type with variable, timer and port instance

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

EXAMPLE 3: Component type with port instance arrays

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

6.2.10.2 Reuse of component types

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

Syntactical Structure

```
type component ComponentTypeIdentifier extends ComponentTypeIdentifier "{"
     { ( PortInstance
     | VarInstance
     | TimerInstance
     | ConstDef ) }
"}"
```

Semantic Description

In such a definition, the new type definition is referred to as the *extended type*, and the type definition following the **extends** keyword is referred to as the *parent type*. The effect of this definition is that the extended type will implicitly also contain all definitions from the parent type. It is called the *effective type definition*.

It is allowed to have one component type extending several parent types in one definition, which have to be specified as a comma-separated list of types in the definition. Any of the parent types may also be defined by means of extension. The effective component type definition of the extended type is obtained as the collection of all constant, variable, timer and port definitions contributed by the parent types (determined recursively if a parent type is also defined by means of an extension) and the definitions declared in the extended type directly. The effective component type definition shall be name clash free.

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

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

NOTE 2: For component type compatibility, this means that a component reference c of type CT1, which extends CT2, is compatible with CT2, and test cases, functions and altsteps specifying CT2 in their **runs on** clauses can be executed on c (see clause 6.3.3).

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) When defining component types by extension, there shall be no name clash between the definitions being taken from the parent type and the definitions being added in the extended type, i.e. there shall not be a port, variable, constant or timer identifier that is declared both in the parent type (directly or by means of extension) and the extended type. It is not considered to be a name clash if a specific definition is contributed to the extended type via different extension paths.

b) When defining component types by extending more than one parent type, there shall be no name clash between the definitions of the different parent types, i.e. there shall not be a port, variable, constant or timer identifier that is declared in any two of the parent types (directly or by means of extension). It is not considered to be a name clash if a specific definition is contributed to the extended type via different extension paths.

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

Examples

EXAMPLE 1: A component type extension and its effective type definition

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

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

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

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

EXAMPLE 2: A component type extension chain and forbidden cyclic extensions

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

EXAMPLE 3: Component type extensions with name clashes

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

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

EXAMPLE 4: Component type extension from several parent types

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

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

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

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

6.2.11 Component references

Component references are unique references to the test components created during the execution of a test case.

Syntactical Structure

system | **mtc** | **self** | *VariableRef* | *FunctionInstance*

Semantic Description

A unique component reference is generated by the test system at the time when a component is created. It is the result of a **create** operation (see clause 21.2.1). In addition, component references are returned by the predefined operations **system** (returns the component reference of the test system interface, which is automatically created when testcase execution is started), **mtc** (returns the component reference of the MTC, which is automatically created when testcase execution started) and **self** (returns the component reference of the component in which **self** is called).

Component references are used in the configuration operations such as **connect**, **map** and **start** (see clause 21) to set-up test configurations and in the **from**, **to** and **sender** parts of communication operations of ports connected to test components other than the test system interface for addressing purposes (see clause 22 and figure 6).

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

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

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

The configuration operations (see clause 21) do not work directly on arrays of components. Instead a specific element of the array shall be provided as the parameter to these operations. For components, the effect of an array is achieved by using an array of component references and assigning the relevant array element to the result of the **create** operation.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The only operations allowed on component references are assignment, equality and non-equality.

b) The variable associated with *VariableRef* (being a component type variable, a component type parameter, etc.) or the return type associated with *FunctionInstance* must be of component type.

Examples

EXAMPLE 1: Component references with component type variables

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

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

EXAMPLE 2: Usage of component references in configuration operations

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

EXAMPLE 3: Usage of component references in from- and to- clauses

```
MyPCO1.receive from MyCompInst;
 :
MyPCO2.receive(integer:?) -> sender MyCompInst;
 :
MyPCO1.receive(MyTemplate) from MyCompInst;
 :
MyPCO2.send(integer:5) to MyCompInst;
```

EXAMPLE 4: Usage of component references in one-to-many connections

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

EXAMPLE 5: Usage of self

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

EXAMPLE 6: Usage of component arrays

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

testcase MyTestCase() runs on MyMtcType system MyTestSystemInterface
{
     :
    var integer i;
    var MyPTCType1  MyPtc[11];
     :
    for (i:= 0; i<=10; i:=i+1)
    {
        MyPtc[i] := MyPTCType1.create;
        connect(self:PtcCoordination, MyPtc[i]:MtcCoordination);
        MyPtc[i].start(MyPtcBehaviour());
    }
     :
}
```

6.2.12 Addressing entities inside the SUT

An SUT may consist of several entities which can be addressed individually. The global **address** data type may be used if only one data type is needed. If several data types at different ports are needed for addressing SUT entities, the type used for addressing via a port instance shall be declared in the corresponding port type definition.

Syntactical Structure

TemplateInstance

Semantic Description

The actual data representation of the global **address** type is resolved either by an explicit global address type definition within the test suite, address type definitions within port definitions, or externally by the test system (i.e. the **address** type is left as an open type within the TTCN-3 specification). This allows abstract test cases to be specified independently of any real address mechanism specific to the SUT.

If an **address** type is bound to a port type definition, addressing of SUT instances (i.e., **to**- and **from**-directives in communication operations) via instances of that port type shall be restricted to values of the bound **address** type.

If several address types exist within a test suite, ambiguities shall be resolved by means of the dot notation. For example, a type reference within a variable definition used to store an SUT address may be prefixed by a port type identifier or a module identifier. If both a global address type definition and port definitions with an address type definition exist in a module, the global address type shall only affect ports without an explicit address type definition. The consistent use of explicit address type definitions within port definitions is recommended over the use of global address type definitions.

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

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

If a port type definition includes the declaration of a type that shall be used for addressing SUT entities, only values of that type shall be used in **to**, **from** and **sender** parts of receive and send operations of port instances of that type mapped to the test system interface.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) *TemplateInstance* shall be of type **address** or of the type of the address declaration in a port type definition. If *TemplateInstance* is of type **address**, it and can be an address type value, an address type variable, etc.

b) For addressing purposes, the **address** data type shall only be used in the **to**, **from** and **sender** parts of receive and send operations of ports mapped to the test system interface.

Examples

EXAMPLE 1: Global address type

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

EXAMPLE 2: Port type-specific address type

```
type record MyAddressType {                          // user-defined type
    integer field1;
    boolean field2;
}
type port MyPortType message {
    address MyAddressType;                       // address declaration
    inout   integer;
}
type component MyComponentType
{
    port MyPortType PCO;
}
function myFunction () runs on MyComponentType {
    var MyAddressType SUT_Address := { 5, true};    // address value for addressing via ports
                                                 // of MyPortType
    :
    PCO.send(integer: 5) to SUT_Address;             // use of address value in to
    :
    PCO.receive(integer: ?) from SUT_Address;    // use of address value in from
    :
}
```

EXAMPLE 3: Elaborated address example

```
type AddressType1 address;              // address type definition on module level

type port MyPortType1 message {
    inout   MsgType1;
}

// address types bound to port types
type port MyPortType2 message {
    address AddressType2;               // values of type AddressType2 can be
                                        // used to address SUT entities.
    inout   MsgType2;
}
type port MyMessagePort3 message {
    address AddressType3;               // values of type AddressType3 can be
                                        // used to address SUT entities.
    inout   MsgType3;
}
// component type definition
type component MyComponentType
{
    port MyPortType1    PCO1;
    port MyPortType2    PCO2;
    port MyPortType3    PCO3
}
// The following behaviour is considered to be executed on an instance of MyComponentType.
// Furthermore, it is considered that the ports PCO1, PCO2 and PCO3 are mapped ports, i.e.,
// used for the communication with the SUT.
 :
```

```
// new address variable initialized with null
var address                 MySUTentity1 := null;   // type of MySUTentity1 is AddressType1
var MyPortType2.address MySUTentity2 := null;   // type of MySUTentity2 is AddressType2
var MyPortType3.address MySUTentity3 := null;   // type of MySUTentity3 is AddressType3
 :
// receiving an address values and assigning them to variables
PCO1.receive(MsgType1:?) from address:? -> sender MySUTentity1;
                                                    // Address type of module scope,
                                                // no prefix needed
PCO2.receive(MsgType2:?) from MyPortType2.address:? -> sender MySUTentity2;
                                                    // Resolution of address type
                                                // by means of a prefix
PCO3.receive(MsgType3:?) from MyPortType3.address:? -> sender MySUTentity3;
 :
// usage of the received address values for addressing purposes
PCO1.send(MyResult) to MySUTentity1;
 :
PCO2.receive(MyConfirmation) from MySUTentity2;
 :
PCO3.send(MyRequest) to MySUTentity3;
 :
```

6.2.13 Subtyping of structured types

TTCN-3 allows subtyping of structured types as given in table 3.

6.2.13.1 Length subtyping of record ofs and set ofs

TTCN-3 permits constraining the number of elements in instances of **record of** and **set of** types.

The **length** keyword followed by a value or a range (with inclusive boundaries only) within brackets and used between the **record** or **set** and the **of** keywords, restricts the allowed lengths of the given **record of** or **set of** type. The value or the bounds within the brackets shall be non-negative integer values, except when the **infinity** keyword is used at the place of the upper bound, in which case the maximum number of the elements is not constrained.

Record of and set of type definitions may be used to define new **record of** or **set of** subtypes. In this case the rules of the previous paragraph apply, except that the **length** keyword and the value or range defining the allowed number of iterations (within brackets) shall be placed following the identifier of the new type.

Constants used in the constant expressions of length subtyping shall meet with the restrictions in clause 10.

EXAMPLE 1: Length restrictions of record of and set of types

```
type record length(10) of integer MyRecordOfType10;
// is a record of exactly 10 integers

type record length(0..10) of integer MyRecordOfType0_10;
// is a record of a maximum of 10 integers

type record length(10..infinity) of integer MyRecordOfType10up;
// record of at least 10 integers

type record length(0..infinity) of integer MyRecordOfType0up;
// an unrestricted record of integer type
```

EXAMPLE 2: Length subtyping of referenced record of types

```
type record of charstring StringArray;
// is an unlimited record of, each element shall be a charstring

type StringArray StringArray34 length(4 .. 5);
// is a record of 4 or 5 elements, each element is a charstring
// it is equivalent to
// type record length(4 .. 5) of charstring StringArray34a;

type StringArray34 StringArray34again length(4 .. 5);
// the same as StringArray34

type StringArray34 StringArray6 length(6);
// causes an error as record ofs with 6 elements are not legal values of StringArray34
```

EXAMPLE 3: Length subtyping of referenced set of types

```
type record MyCapsule {
  set of integer mySetOfInt
}

type MyCapsule.mySetOfInt MySetOfIntSub length(5..10);
// unordered list of 5 to 10 integers
```

6.2.13.2 List subtyping of structured types and anytype

List subtyping is possible when defining a new type based on an existing parent type, but not directly at the declaration of the first parent type (see table 3).

Subtypes defined by a list subtyping restrict the allowed values of the subtype to the values in the list. In case of list subtyping of **record**, **set**, **record of**, **set of**, **union** and **anytype** types, the list may contain both values and subtypes of the parent types of the type being constrained. The collection of values denoted by the type(s) referenced in the list become instances of the new subtype. All values of the expanded list (i.e. after resolving the type references) shall be valid values of the first parent type.

In case of **enumerated** types, the value list subtyping shall contain only values of the parent type.

EXAMPLE 1: List subtyping of record types

```
type record MyRecord {
  integer      f1  optional,
  charstring   f2,
  charstring   f3
}

type MyRecord MyRecordSub1 (
  { f1 := omit, f2 := "user", f3 := "password" },
  { f1 := 1, f2 := "User", f3 := "Password" }
) // a valid subtype of MyRecord containing 2 values

type MyRecord MyRecordSub2 (
  MyRecordSub1,
  { f1 := 2, f2 := "uname", f3 := "pswd" },
  { f1 := 3, f2 := "Uname", f3 := "Pswd" }
) // a valid subtype MyRecord containing 4 values; notice that values of
  // MyRecordSub1 are identified by referencing MyRecordSub1

type MyRecordSub1 MyRecordSub3 (
  { f1 := 1, f2 := "user", f3 := "password" },
  { f1 := 1, f2 := "User", f3 := "Password" }
) // empty type as { f1 := 1, f2 := "user", f3 := "password" } is not a legal value of
  // MyRecordSub1 (notice field f1)
```

EXAMPLE 2: List subtyping of record of types

```
type record of charstring StringArray;

type StringArray StringArrayList1 (
  { "aa" },
  { "bbb", "cc" },
  { "ddd", "ee", "ff" }
); // valid subtype of StringArray

type StringArrayList1 StringArrayList2 (
  { "aa" },
  { "bbb", "cc" }
); // valid subtype of StringArrayList1

type StringArrayList1 StringArrayList3 (
  StringArrayList2,
  { "ddd", "ee", "ff" }
); // valid, but equivalent to StringArrayList1

type StringArrayList1 StringArrayList4 (
  StringArrayList2,
  { "ddd", "ee", "fff" }
); // empty type as { "ddd", "ee", "fff" } is not a value of StringArrayList1
   // (notice the extra character f in the third element)
```

EXAMPLE 3: List subtyping of union types

```
type union MyUnion {
  integer       c1,
  charstring    c2,
  charstring    c3
};

type MyUnion MyUnionSub1 (
  { c1 := 0 },
  { c1 := 1 }
); // a valid subtype of MyUnion containing two values

type MyUnion MyUnionSub2 (
  MyUnionSub1,
  { c2 := "mine" },
  { c3 := "yours" }
); // a valid subtype of MyUnion containing four values; notice that values of
   // MyUnionSub1 are identified by referencing MyUnionSub1

type MyUnionSub1 MyUnionSub3 (
  { c1 := 0 },
  { c1 := 2 }
); // causes an error as { c1 := 2 } is not a value of MyUnionSub1
```

EXAMPLE 4: List subtyping of enumerated types

```
type enumerated MyEnum { first, second, third, fourth, fifth };

type MyEnum EnumSub1 ( first, second, third );
  // a valid subtype of MyEnum

type EnumSub1 EnumSub2 ( first, second );
  // a valid subtype of EnumSub1

type EnumSub1 EnumSub3 ( first, second, fourth );
  // causes an error as fourth is not a value of EnumSub1

type MyEnum EnumSub4 ( EnumSub1, fourth);
  // causes an error as type references are not allowed in the value list of enumerated types
```

EXAMPLE 5: List subtyping of anytype

```
type anytype MyAnySub1 (
  { integer := 5 },
  { boolean := false },
  { bitstring := '0011'B },
  { charstring := "mine" },
  { MyEnum := first }
); // a valid subtype of anytype, consisting of 5 values

type MyAnySub1 MyAnySub2 (
  { integer := 5 },
  { boolean := false },
  { bitstring := '0011'B }
); // a valid subtype of MyAnySub1, consisting of 3 values

type anytype MyAnySub3 (
  MyAnySub2,
  { octetstring := 'FF'O }
); // a valid subtype of anytype, consisting of 4 values, 3 of which are defined
   // by referring to MyAnySub2

type MyAnySub1 MyAnySub4 (
  { integer := 5 },
  { boolean := false },
  { MyEnum := second }
); // causes an error as { MyEnum := second } is not a value of MyAnySub1

type MyAnySub1 MyAnySub5 (
  MyAnySub3,
  { MyEnum := first }
); // causes an error as { octetstring := 'FF'O } (defined via referencing MyAnySub3) is
   // not a value of MyAnySub1
```

6.2.13.3 Subtyping of the iterated type of record ofs and set ofs

A type restriction following the identifier of a newly defined **record of** or **set of** type (i.e. when the keywords **record** and **of** or **set** and **of** are used in the definition) shall constrain the innermost type. The newly defined iterated type shall be a subset of the innermost type. If the innermost type is a basic type, the subtyping rules in clause 6.1.2 shall apply. If the innermost type is referencing a structured type or **anytype**, the rules in clauses 6.2.13.1 and 6.2.13.2 shall apply.

EXAMPLE 1: Subtyping of basic innermost types of record ofs and set ofs

```
type record of charstring String23Array length(2 .. 3);
  // is an unlimited record of, each element shall be a charstring of 2 or 3 characters

type record length(0..10) of charstring String12Array10 length(12);
// is a record of a maximum of 10 strings each with exactly 12 characters

type record of record of charstring String12Array2D length(12);
// is a two-dimensional unlimited array of strings each with exactly 12 characters

type set length(5) of set length(6) of charstring String12Array2D56 length(12);
// is an unordered two-dimensional array of the size 5*6 strings, each with
// exactly 12 characters


const String23Array c_str23arr_a := { "aa", "bbb", "cc", "ddd", "ee", "ff" };
// valid, all charstrings are 2 or 3 characters long

const String23Array c_str23arr_b := { "a", "bbbb", "cc", "ddd", "ee", "ff" };
// causes an error as "a" and "bbbb" are not 2 or 3 characters long

const String12Array2D56 c_str12arr2D56_a := {
  { "aa", "aaa", "bb", "bbb", "cc", "ccc" },
  { "dd", "ddd", "ee", "eee", "ff", "fff" },
  { "gg", "ggg", "hh", "hhh", "ii", "iii" },
  { "jj", "jjj", "kk", "kkk", "ll", "lll" },
  { "mm", "mmm", "nn", "nnn", "oo", "ooo" }
}; // valid, a 5*6 matrix of charstrings being 2 or 3 characters long

const String12Array2D56 c_str12arr2D56_b := {
  { "a", "aaa", "bb", "bbbb", "cc", "ccc" },
  { "dd", "ddd", "ee", "eee", "ff", "fff" },
  { "gg", "ggg", "hh", "hhh", "ii", "iii" },
  { "jj", "jjj", "kk", "kkk", "ll", "lll" },
  { "mm", "mmm", "nn", "nnn", "oo", "ooo", "pp" }
}; // causes an error as "a" and "bbbb" are not 2 or 3 characters long and
   // the 5th inner record of has 7 elements
```

EXAMPLE 2: Length subtyping of structured innermost types of record ofs

```
type record of String23Array String23Array45 length(4 .. 5);
// is a two-dimensional array, the first dimension is unlimited,
// the second dimension is restricted to 4 or 5 elements and each element
// is a charstring of 2 or 3 characters. It is equivalent to:
// type record of record length(4 .. 5) of charstring String23Array45 length(2 .. 3);

const String23Array45 c_str23arr45_a := {
  { "aa", "bbb", "cc", "ddd" },
  { "ee", "fff", "gg", "hhh", "ii" }
}; // valid, 4 or 5 elements in the inner record of, all containing 2 or 3 characters

const String23Array45 c_str23arr45_b := {
  { "aa" , "bbb", "cc" }
}; //causes an error as there are only 3 elements in the inner record of

const String23Array45 c_str23arr45_c := {
  { "aa", "bbbb", "cc", "dd" }
}; //causes an error as "bbbb" contains 4 characters

type record length(0 .. 1) of String23Array String23Array0145 length(4 .. 5);
// is a two-dimensional array, the first dimension is limited to 0 or 1 elements,
// the second dimension is restricted to 4 or 5 elements, each element is a
// charstring of 2 or 3 characters.

const String23Array0145 c_str23arr0145_a := {
  { "aa", "bbb", "cc", "ddd" },
}; // a valid 1*4 array of charstrings, each of 2 or 3 characters
```

```
const String23Array0145 c_str23arr0145_a := {
  { "aa", "bbb", "cc", "ddd" },
  { "ee", "fff", "gg", "hhh", "ii" }
}; // causes an error as there are two elements in the outer record of

const String23Array0145 c_str23arr0145_b := {
  { "aa" , "bbb", "cc" }
}; // causes an error as there are only 3 elements in the inner record of

const String23Array0145 c_str23arr0145_c := {
  { "aa", "bbbb", "cc", "dd" }
}; // causes an error as "bbbb" contains 4 characters

type record of String23Array45 String23Array6 length(6);
// empty type as String23Array45 is restricted to 4 or 5 elements,
// thus length restriction 6 is outside the allowed range
```

6.2.13.4 Mixing subtyping mechanisms

In the case of structured types and the special type **anytype**, it is forbidden to mix different subtyping mechanisms (e.g. list and length) in the same definition.

6.3 Type compatibility

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

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

NOTE: As **address** is more a predefined type name than a distinct type with its own properties, the same type compatibility rules apply to an **address** type and to its derivatives as the rules were if the type was defined with a name different from **address**.

6.3.1 Type compatibility of non-structured types

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

EXAMPLE 1: Compatibility of integers

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

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

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

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

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

EXAMPLE 2: Compatibility of floats

```
// Given
type float PositiveFloats(0.0 .. infinity);
:
var PositiveFloats x;
var float y;
```

```
// Then
y := 5.0; // is a valid assignment
x := y;
// is a valid assignment, because y has the same root type as x and no subtyping is violated

y := -20.0; // is a valid assignment
x := y;
// causes an error, because the value of y is out of the range of PositiveFloats

y := not_a_number; // is a valid assignment
x := y;
// causes an error, because the value not_a_number is out of the range of PositiveFloats
```

EXAMPLE 3: Compatibility of charstrings

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

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

//Given
myCharString := "abcd";

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

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

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

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

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

6.3.2 Type compatibility of structured types

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

6.3.2.1 Type compatibility of enumerated types

Enumerated types are only compatible to synonym types (see clause 6.4) and not compatible with other basic or structured types.

6.3.2.2 Type compatibility of record and record of types

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

EXAMPLE 1:

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

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

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

}

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

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

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

// Then

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

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

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

Two array types are compatible if their corresponding **record of** types are compatible.

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

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

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

EXAMPLE 2:

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

type record of integer IType

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

// Then

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

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

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

6.3.2.3 Type compatibility of set and set of types

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

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

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

EXAMPLE:

```
// Given
type set FType {
    integer a   optional,
    integer b   optional,
    boolean c
}

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

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

// Then

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

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

6.3.2.4 Type compatibility of union types

union types are only type compatible with other **union** types. A union value "a" of union type "A" is compatible with union type "B" if the alternative selected in "a" has a corresponding alternative with identical name in "B" and the value of the selected alternative in "a" is compatible to the type of the corresponding alternative in "B".

EXAMPLE:

```
type union U1 {integer i};
type union U2 {integer i, boolean b};

var U1 u1 := {i := 1};
var U2 u2 := u1;        // correct
u1:= u2;                // correct as the alternative i is selected in u2 and is compatible
                        // to i in U1
u2:= {b := true};
u1:= u2;                // incorrect as u1 has no alternative b
var anytype x := u1;    // incorrect as the anytype is not a union type.
```

6.3.2.5 Type compatibility of anytype types

anytype types are only type compatible with other **anytype** types. An anytype value "a" of anytype type "A" is compatible with anytype type "B" if the alternative selected in "a" has a corresponding alternative with identical name in "B" and the value of the selected alternative in "a" is compatible to the type of the corresponding alternative in "B". Identical alternative names in this case mean the name of a TTCN-3 basic type or the name of the same user defined type definition (considering also the module in which the type is defined).

EXAMPLE:

```
module A {
    type integer I (0..2);
    type float F;
    type anytype Atype //anytype composed of TTCN-3 built-in basic types, I, and F
}

module B {
    type integer I (0..2);
    type anytype Atype
}

module C {
    import from A all;
    import from B all;
    type union U {
        integer I (0..2)
    }
    control {
        var A.Atype aa;
        var A.Atype aaI := { I := 1 }
        var A.Atype aaF := { F := 1.0 }
        var B.Atype ba := { integer := 1 }
        var B.Atype baI := { I := 1 }
        var U u := { I := 1 }

    aa := ba;       // correct, the value of aa1 becomes { integer := 1 }
    aa := baI;      // incorrect, type B.I is not present in the anytype A.Atype
    aa := u;        // incorrect, type of u is not anytype but a user defined union type

    ba := { float := 1.0 }; // correct, assigning a literal value
    ba := aaI;      // incorrect, type A.I is not present in the anytype B.Atype
    ba := aaF;      // incorrect, type A.F is not present in the anytype B.Atype
    }
}
```

6.3.2.6 Compatibility between sub-structures

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

EXAMPLE:

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

var JType MyVarJ

// If considering the declarations above, then


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

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

6.3.3 Type compatibility of component types

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

1) Compatibility of a component reference value with a component type (e.g. when passing a component reference as an actual parameter to a function or an altstep or when assigning a component reference value to a variable of different component type): a component reference "b" of component type "B" is compatible with component type "A" if all definitions of "A" have identical definitions in "B".

2) Runs on compatibility: a function or altsteps referring to component type "A" in its runs on clause may be called or started on a component instance of type 'B' if all the definitions of "A" have identical definitions in "B".

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

a) For port instances, both the type and the identifier shall be identical.

b) For timer instances, identifiers shall be identical and either both shall have identical initial durations or both shall have no initial duration.

c) For variable instances and constant definitions, the identifiers, the types and initialization values shall be identical (in case of variables this means that either the values are missing in both definitions or are the same).

d) For local template definitions, the identifiers, the types, the formal parameter lists and the assigned template or template field values shall be identical.

6.3.4 Type compatibility of communication operations

The communication operations (see clause 22) **`send`**, **`receive`**, **`trigger`**, **`call`**, **`getcall`**, **`reply`**, **`getreply`** and **`raise`** are exceptions to the weaker rule of type compatibility and require strong typing. The types of values or templates directly used as parameters to these operations must also be explicitly defined in the associated port type definition. Strong typing also applies to storing the received value, address or component reference during a **`receive`** or **`trigger`** operation.

EXAMPLE:

```
type record MyRec {...}                          // user defined type
type MyRec MyRecAlias;                           // a type alias

type port MyPort message { inout MyRec, MyRecAlias; }    // port that can transport both types
type component MyComponent { port MyPort P; }
```

```
template MyRecAlias t_MyRecAlias:= {...}          // a template of the alias type

var MyComponent myComp1 := MyComponent.create, myComp2 := MyComponent.create;
connect (myComp1:P, myComp2:P)                    // two connected PTCs via ports that can
                                                  // transport the user-defined and the alias type

// in myComp1:
P.send (t_MyRecAlias);                            // sending of template of alias type

// in myComp2:
P.receive (MyRec:?);
// shall not match as the transmitted template is of the alias type and
// not of the user-defined type

// in myComp2:
var MyRec x;
P.receive (MyRecAlias:?) -> value x;
// shall cause an error since also storing the value requires strong typing
```

6.3.5 Type conversion

If it is necessary to convert values of one type to values of another type, because their types have different root types, then either one of the predefined conversion functions defined in clause 16.1.2 or a user defined function shall be used.

EXAMPLE:

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

6.4 Type synonym

A type can be defined as a synonym to another type. Type synonyms can be defined for all kinds of types. Synonym types are compatible.

EXAMPLE:

```
type MyType1 MyType2; // MyType2 is synonym to MyType1
```

7 Expressions

TTCN-3 allows the specification of expressions using the operators defined in clause 7.1.

Syntactical Structure

```
SingleExpression |
"{" { ( FieldReference ":=" ( Expression | "-" )) [","] } "}" |  // compound expression
"{" [ { ( Expression | "-" ) [","] } ] "}"                       // compound expression
```

Semantic Description

Expressions may be built from other (simple) expressions. Functions used in expressions shall have a return clause. The operands of the operators used in an expression shall be values and their root types shall be the types specified for the appropriate operator in the subsequent clauses.

Compound expressions are used for expressions of array, record, record of and set of types.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) Operands of operators used in expressions shall be completely initialized.

b) The root types of the operands shall be the types specified for the appropriate operand.

This means also that all fields and elements of structured types referenced in an expression shall contain completely initialized values, while other fields and elements, not used in the expression, may be uninitialized or contain **omit**.

Examples

```
(x + y - increment(z))*3            // single expression
{ a:= 1, b:= true }                 // compound expression, field expression list
{ 1, true }                         // compound expression, value list
```

7.1 Operators

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

a) arithmetic operators;

b) list operator;

c) relational operators;

d) logical operators;

e) bitwise operators;

f) shift operators;

g) rotate operators.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) Values used in operators shall be completely initialized.

These operators are listed in table 5.

Table 5: List of TTCN-3 operators

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

The precedence of these operators is shown in table 6. Within any row in this table, the listed operators have equal precedence. If more than one operator of equal precedence appears in an expression, the operations are evaluated from left to right. Parentheses may be used to group operands in expressions, in which case a parenthesized expression has the highest precedence for evaluation.

Table 6: Precedence of Operators

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

7.1.1 Arithmetic operators

The arithmetic operators represent the operations of addition, subtraction, multiplication, division, modulo and remainder. Operands of these operators shall be of **integer** values (including derivations of **integer**) or floating-point numbers (including derivations of **float**, containing numeric values only), except for **mod** and **rem** which shall be used with **integer** (including derivations of **integer**) types only.

NOTE: The special float values **infinity**, **-infinity** and **not_a_number** are not to be used with arithmetic operators.

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

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

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

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

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

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

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

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

Table 7 illustrates the difference between the **mod** and **rem** operator:

Table 7: Effect of mod and rem operator

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

7.1.2 List operator

The predefined list operator (&) performs concatenation of values of string types, **record of**, **set of**, or **array** of the same root types. The operation is a simple concatenation from left to right. No form of arithmetic addition is implied. The result type is the root type of the operands.

NOTE: In case of the list types, both the outer type (i.e. **record of**, **set of** or **array**) and the iterated inner type need to have the same root type in a recursive manner.

EXAMPLE:

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

7.1.3 Relational operators

The predefined relational operators are equality (==), less than (<), greater than (>), non-equality to (!=), greater than or equal to (>=) and less than or equal to (<=). The result type of all these operations is **boolean**.

The relational operators less than (<), greater than (>), greater than or equal to (>=), and less than or equal to (<=) shall have only operands of type **integer** (including derivations of **integer**), **float** (including derivations of **float**), or instances of the same **enumerated** type. It is not allowed to compare instances of different root types.

Operands of equality (==) and non-equality (!=) shall be values of the same root type and the values being compared shall obey the following rules. This implies that instances of types not mentioned below shall not be operands of equality and non-equality.

NOTE: As **address** is more a predefined type name than a distinct type with its own properties, the same rules apply to an **address** type and to its derivatives as the rules were if the type was defined with a name different from **address**.

- Two integer values are equal if and only if they contain the same value. Otherwise, normal mathematical ordering is applied.
- Two floating-point numbers are equal if and only if they contain the same value. The values minus zero and plus zero are two distinct values (e.g. they are encoded differently in some standardized languages) and minus zero is less than plus zero, which represents zero. Otherwise, normal mathematical ordering is applied. The special values **-infinity, infinity** and **not_a_number** are equal to themselves only. The special value **-infinity** is less than any other foat value. The special value **infinity** is greater than any numerical float values and **-infinity**. The special value **not_a_number** is greater than any other float value (including **infinity)**.
- Two charstring or two universal charstring values are equal if and only if they have equal lengths and the characters at all positions are the same.
- For values of bitstring, hexstring or octetstring types, the same equality rule applies as for charstring values with the exception, that fractions which shall equal at all positions are bits, hexadecimal digits or pairs of hexadecimal digits accordingly.
- Two record values, set values, record of values or set of values are equal if and only if their effective value structures are compatible (see clause 6.3) and the actual values of all corresponding fields are equal. record values may also be compared to record of values and set values to set of values. In these cases the same rule applies as for comparing two **record** or **set** values.

- Values of the same union type, and values of different union types in which at least one of the alternatives is compatible with the other type (see clause 6.3.2.4) can be compared (independent if a compatible alternative is the selected one or not). Two values of union types are equal if and only if in both values the name of the selected alternative is identical, they are compatible with the type of the other value, and the actual values of the chosen fields are equal.

- Values of the same or any two anytype types can be compared. For anytype values the same rule apply as to union values, with the addition that names of types defined with the same name in different modules do not denote the same name of the selected alternatives.

- Two default or two component values are equal if and only if they contain the same value (i.e. they designate the same default or test component, independent of the actual state of the denoted object).

EXAMPLE:

```
// Given
type    set S1  {
            integer a1  optional,
            integer a2  optional,
            integer a3  optional
            };

type    set S2  {
            integer b1  optional,
            integer b2  optional,
            integer b3  optional
            };

type    set S3  {
            integer c1  optional,
            integer c2  optional,
            };

type    set of integer  SI;

type    union   U1  {
            integer d1,
            integer d2,
            };

type    union   U2  {
            integer e1,
            integer e2,
            };

type    union   U3  {
            integer d1,
            integer d2,
            boolean d3
            };

// And
const   S1  s1  :=  { a1 := 0, a2 := omit, a3 := 2 };
    // Notice that the order of defining values of the fields does not matter
const   S2  s2a :=  { b1 := 0, b3 := 2, b2 := omit };
const   S2  s2b :=  { b2 := 0, b3 := 2, b1 := omit };
const   S3  s3  :=  { c1 := 0, c2 :=2 };
var     SI  v_si:=  { 0, -, 2 };
const   SI  si  :=  { 0, 2 };
const   U1  u1  :=  { d1:= 0 };
const   U2  u2  :=  { e1:= 0 };
const   U3  u3; :=  { d1:= 0 };

// Then
s1 == s2a;
    // returns true
s1 == s2b;
    // returns false, because neither a1 nor a2 are equal to their counterparts
    // (the corresponding element is not omitted)
s1 == s3;
    // returns false, because the effective value structures of s1 and s3 are not compatible
s1 == v_si;
    // causes test case error as v_si is not completely initialized
    // (2nd element is left uninitialized)
s1 == si;
```

```
    // returns false, as the counterpart of the omitted a2 is 2,
    // but the counterpart of a3 is undefined
s3 == si;
    // returns true
u1 == u2;
    // causes error as U1 and U2 have no common subset of alternatives
u1 == u3;
    // returns true, as alternatives with the same names are chosen and
    // the actual values in the selected alternatives are equal
```

7.1.4 Logical operators

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

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

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

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

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

Short circuit evaluation for boolean expressions is used, i.e. the evaluation of operands of logical operators is stopped once the overall result is known: in the case of the **and** operator, if the left argument evaluates to **false**, then the right argument is not evaluated and the whole expression evaluates to **false**. In the case of the **or** operator, if the left argument evaluates to **true**, then the right argument is not evaluated and the whole expression evaluates to **true**.

7.1.5 Bitwise operators

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

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

Their operands shall be of root type **bitstring, hexstring** or **octetstring**. In the case of **and4b, or4b** and **xor4b** the operands shall be of the same root types.The result type of the bitwise operators shall be the root type of the operands.

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

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

EXAMPLE 1:

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

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

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

EXAMPLE 2:

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

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

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

EXAMPLE 3:

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

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

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

EXAMPLE 4:

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

7.1.6 Shift operators

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

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

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

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

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

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

EXAMPLE 1:

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

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

EXAMPLE 2:

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

7.1.7 Rotate operators

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

NOTE: Please note that the root types of arrays is **record of**, therefore arrays are allowed as left-hand operands of rotate operators.

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

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

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

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

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

e) **record of, set of,** or **array** then the rotate unit applied is one element.

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

EXAMPLE 1:

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

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

EXAMPLE 2:

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

7.2 Field references and list elements

Within expressions, fields of record and set types are referenced with the dot notation **".field"**. Elements of record of, set of, array and string types are referenced with the index notation **"[index]"**. Dot and brackets have the same binding power. Field references and list elements are evaluated from left to right.

8 Modules

The principal building blocks of TTCN-3 are modules. A module may define a fully executable test suite or just a library. A module may refer to the TTCN-3 language version and to package versions being used. A module consists of a (optional) definitions part, and a (optional) module control part.

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

The transfer syntax of TTCN-3 modules shall be UTF-8, i.e. each character of the module shall be individually encoded and decoded according to the UCS Transformation Format 8 (UTF-8) as defined in annex R of ISO/IEC 10646 [2] and no characters not corresponding to any character of the module shall be present.

8.1 Definition of a module

A module is defined with the keyword **module**.

NOTE 1: The treatment of TTCN-3 modules in files, repositories and alike is outside the scope of the present document.

Syntactical Structure

```
module ModuleIdentifier [ language FreeText { "," FreeText } ] "{"
     [ ModuleDefinitionsPart ]
     [ ModuleControlPart ]
"}"
```

Semantic Description

A TTCN-3 module groups a set of (typically cohesive) TTCN-3 definitions. TTCN-3 modules have an explicit import interface to use definitions from other TTCN-3 or non-TTCN-3 modules. It is possible to hide definitions in a TTCN-3 module (see clause 8.2.5). TTCN-3 modules can be compiled/interpreted separately. They are reusable and parameterizable.

Module names are of the form of a TTCN-3 identifier.

NOTE 2: The module identifier is the informal text name of the module.

In addition, a module specification can carry an optional attribute identified by the **`language`** keyword that identifies the edition of the TTCN-3 language, in which the module is specified. The following language strings are to be used:

`"TTCN-3:2001"` - to be used with modules complying with version 1.1.2 of the present document (see annex H).
`"TTCN-3:2003"` - to be used with modules complying with version 2.2.1 of the present document (see annex H).
`"TTCN-3:2005"` - to be used with modules complying with version 3.1.1 of the present document (see annex H).
`"TTCN-3:2007"` - to be used with modules complying with version 3.2.1 of the present document (see annex H).
`"TTCN-3:2008"` - to be used with modules complying with version 3.3.2 of the present document (see annex H).
`"TTCN-3:2008 Amendment 1"` - to be used with modules complying with version 3.4.1 of the present document (see annex H).
`"TTCN-3:2009"` - to be used with modules complying with version 4.1.1 of the present document (see annex H).
`"TTCN-3:2010"` - to be used with modules complying with version 4.2.1 of the present document (see annex H).
`"TTCN-3:2011"` - to be used with modules complying with the present document.

Furthermore, the optional attribute identified by the **`language`** keyword may identify package versions being used by this module. The package tags are defined in ES 202 781 [i.11], ES 202 782 [i.14], ES 202 784 [i.12], and ES 202 785 [i.13]. The language identifier and the package identifier are to be written as a comma-separated list.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

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

8.2 Module definitions part

The module definitions part specifies the top-level definitions of the module and may import visible identifiers from other modules. Visibility rules are given in clause 8.2.5. Scope rules for declarations made in the module definitions part and imported declarations are given in clause 5.3. Those language elements which may be defined in a TTCN-3 module are listed in table 1. Every definition can be associated with attributes using the with statement defined in clause 27. Visible module definitions may be imported by other modules.

Syntactical Structure

```
{
    [ Visibility ] (
        TypeDef |
        ConstDef |
        TemplateDef |
        ModuleParDef |
        FunctionDef |
        SignatureDef |
        TestcaseDef |
        AltstepDef |
        ImportDef |
        GroupDef |
        ExtFunctionDef |
        FriendDef
    ) [ WithStatement ]
    [ ";" ]
}+
```

Semantic Description

Definitions in the module definitions part may be made in any order.

Such definitions, i.e. top level definitions outside of other scope units, are globally visible within the module. They may be used elsewhere in the module. This includes identifiers imported from other modules.

Declarations of dynamic language elements such as variables or timers shall only be made in the control part, test cases, functions, altsteps or component types.

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

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

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

8.2.1 Module parameters

Module parameters define a set of values that are supplied by the test environment at run-time. Module parameters do not change their value during test execution. They can be used on right hand side of assignments, in expressions, in actual parameters, and in template definitions, but not within type definitions.

Syntactical Structure

Single type, single module parameter form:

```
[ Visibility ] modulepar ModuleParType ModuleParIdentifier [ ":=" ConstantExpression ] ";"
```

Single type, multiple module parameter form:

```
[ Visibility ] modulepar ModuleParType
     { ModuleParIdentifier [ ":=" ConstantExpression ] "," }
     ModuleParIdentifier [ ":=" ConstantExpression ] ";"
```

Semantic Description

Module parameters behave as global constants at run-time.

Module parameters allow to customize a TTCN-3 test suite for a specific IUT, test setup or test campaign. Module parameters are declared by specifying the type and listing their identifiers following the keyword **`modulepar`**.

It is allowed to specify default values for module parameters. This shall be done by an assignment in the module parameter list. A default value can merely be assigned at the place of the declaration of the module parameter.

If the test system does not provide an actual run-time value for a module parameter, the default value shall be used during test execution, otherwise the actual value provided by the test system. Actual run-time values shall be literals only.

Visible module parameters can be imported.

Optional fields of record and set module parameters or module parameter fields can be initialized explicitly or implicitly. For implicit initialization of the optional fields of a module parameter or a module parameter field, an **`optional`** attribute with the value "**`implicit omit`**" (see clause 27.7) shall be associated with it either directly or via the attribute distribution (scoping) mechanism (see clause 27.1.1).

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) During test execution these values shall be treated as constants.

b) Module parameters shall not be of port type, default type or component type.

c) A module parameter shall only be of type address if the address type is explicitly defined within the associated module.

d) Module parameters shall be declared within the module definition part only.

e) More than one occurrence of module parameters declaration is allowed but each parameter shall be declared only once (i.e. redefinition of the module parameter is not allowed).

f) The constant expression for the default value of a module parameter shall respect the limitations given in clause 16.1.4.

g) Module parameters shall not be used in type or array definitions.

Examples

```
module MyTestSuiteWithParameters
{
    // single type, single module parameter, which is per default public
    modulepar boolean TS_Par0 := true;

    // single type, multiple module parameters with an explicit public visibility
    public modulepar integer TS_Par1, TS_Par2 := 1 + char2int("a");

    ...
}
```

8.2.2 Groups of definitions

In the module definitions part, definitions can be collected in named groups. Grouping is done to aid readability and to add logical structure to the module if required. If necessary, the dot notation shall be used to identify sub-groups within the group hierarchy uniquely, e.g. for the import of a specific sub-group.

Syntactical Structure

```
[ public ] group GroupIdentifier "{"
    { ModuleDefinition [ ";" ] }
"}"
```

Semantic Description

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

Groups and nested groups have no scoping. Please note however, attributes given to a group by an associated with statement apply to all elements of a group (see clause 27). Import statements may import groups so that all visible elements of a group are imported (see clause 8.2.3.3).

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) Group identifiers across the whole module need not necessarily be unique. However, top-level group identifiers and all group identifiers of subgroups of a single group shall be unique.

b) Only **`public`** visibility can be defined for groups as they are always public.

Examples

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

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

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

8.2.3 Importing from modules

It is possible to re-use visible definitions specified in different modules using the **`import`** statement. Every definition in a TTCN-3 module has an associated visibility, which is by default **`public`** (see clause 8.2.5).

NOTE: Groups are **`public`** only. Importing a group means that only the visible elements of the group are being imported.

8.2.3.1 General format of import

An import statement can be used anywhere in the module definitions part.

Syntactical Structure

```
[ Visibility ] import from ModuleId
     (
          ( all [ except "{" ExceptSpec "}" ] )
          |
          ( "{" ImportSpec "}" )
     )
[ ";" ]
```

Semantic Description

TTCN-3 supports the import of the following definitions: module parameters, user defined types, signatures, constants, data templates, signature templates, functions, external functions, altsteps and test cases. Each definition has a *name* (defines the identifier of the definition, e.g. a function name), a *specification* (e.g. a type specification or a signature of a function) and in the case of functions, altsteps and test cases an associated *behaviour description*. In addition, import statements of one module can be explicitly imported by another module (see clause 8.2.3.7). Only definitions or import statements visible from the importing module can be imported (see clause 8.2.5).

In contrast to module definitions, which are by default public, import statements are by default private.

EXAMPLE 1a:

	Name	Specification	Behaviour description

function	MyFunction	(**inout** MyType1 MyPar) **return** MyType2
		runs on MyCompType
			{
			  const MyType3 MyConst := …;
			  : // further behaviour
			}

	Specification	Name	Specification

type	**record**	MyRecordType	{
			  field1 MyType4,
			  field2 integer
			}

	Specification	Name	Specification

template	MyType5	MyTemplate	:= {
			  field1 := 1,
			  field2 := MyConst, // MyConst is a module constant
			  field3 := ModulePar // ModulePar is module parameter
			}

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

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

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

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

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

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

The TTCN-3 import mechanism is related to the local and referenced definitions used in the specification part of the importable definitions. Table 8 specifies the possible local and referenced definitions of importable definitions.

Table 8: Possible local and referenced definitions of importable definitions

Importable Definition	Possible Local Definitions	Possible Referenced Definitions
Module parameter		Module parameter type
User-defined type (for all)		
• enumerated type	Concrete values	
• structured type	Field names, nested type definitions	Field types
• port type		Message types, signatures
• component type	Constant names, variable names, timer names and port names	Constant types, variable types, port types
Signature	Parameter names	Parameter types, return type, types of exceptions
Constant		Constant type
Data Template	Parameter names	Template type, parameter types, constants, module parameters, functions
Signature template		Signature definition, constants, module parameters functions
Function	Parameter names	Parameter types, return type, component type (**runs on**-clause)
External function	Parameter names	Parameter types, return type
Altstep	Parameter names	Parameter types, component type (**runs on**-clause)
Test case	Parameter names	Parameter types, component types (**runs on**- and **system**- clause)
NOTE 1: For the import of import statements see clause 8.2.3.7. NOTE 2: For the import of groups see clause 8.2.3.3.		

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

EXAMPLE 1b: Differentiation between *information necessary for the usage* and the identifier.

```
module A {
  type record MyRec1 {
    integer     field1,
    charstring  field2
  }
}
```

```
module B {
  import from A all;
  type record MyRec2 {
    MyRec1  myField1,
     // "myField1" is the local definition, "MyRec1" is a referenced definition;
     // the name "MyRec1" shall be imported in this case as is directly referenced
    boolean myField2
  }
}

module C {
  import from B all;
  const MyRec2 t_MyRec2 := {
    myField1 := { field1 := 5, field2 := "A" },
      // to define myField1 of MyRec2 the name "MyRec1" is not needed, the
      // information necessary for the usage is its type information,
      // i.e. names and types of its fields field1 and field2
      // which is embeddded in the imported definition of MyRec2
    myField2 := true
  }
}
```

If an imported definition has attributes (defined by means of a **with** statement) then the attributes shall also be imported. The mechanism to change attributes of imported definitions is explained in clause 27.1.3.

NOTE 3: If the module has global attributes they are associated to definitions without these attributes.

The use of **import** on single definitions, groups of definitions, definitions of the same kind, etc. may lead to situations where the *same definition is referred to more than once*. Such cases shall be resolved by the system and definitions shall be imported only once.

NOTE 4: The mechanisms to resolve such ambiguities, e.g. overwriting and sending warnings to the user, are outside the scope of the present document and should be provided by TTCN-3 tools.

All **import** statements and definitions within import statements are considered to be treated independently one after the other in the order of their appearance.

All TTCN-3 modules shall have their own name space in which all definitions shall be uniquely identified. *Name clashes* may occur due to import, e.g. import from different modules. Name clashes shall be resolved using qualified name(s) for the imported definition(s), i.e. prefixing the imported definition (which causes the name clash) by the identifier of the module in which it has been defined; the prefix and the identifier shall be separated by a dot (".").

There is one exception to this rule: when **in the context** of an enumerated type (see clause 6.2.4), an enumeration value is clashing with the name of a definition in the importing module, the enumeration value shall take precedence and the definition in the importing module shall be referenced by using its qualified name (see example 4 below in this clause).

In cases where there are no ambiguities the prefixing need not (but may) be present when the imported definitions are used. When the definition is referenced in the same module where it is defined, the module identifier of the module (the current module) also may be used for prefixing the identifier of the definition.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) An import statement shall only be used in the module definitions part and not be used within a control part, function definition, and alike.

b) Only top-level visible definitions of a module may be imported. Definitions which are top-level but invisible to the importing module or which occur at a lower scope (e.g. local constants defined in a function) shall not be imported.

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

NOTE 5: A local definition, e.g. a field name of a user-defined record type or the name of an enumeration value, has only meaning in the context of the definitions in which it is defined, e.g. a field name of a record type can only be used to access a field of the record type and not outside this context.

In particular, importing an enumerated type does not impose the restriction given in clause 6.2.4 on global names defined in the importing module.

d) A definition is imported together with all information of referenced definitions that are necessary for the usage of the imported definition, independent of the visibility of the referenced definitions (see clause 8.2.5).

NOTE 6: If module C imports a definition from module B that uses a type reference defined in module A, the corresponding information necessary for the usage of that type is automatically imported into module C (see example 5 below in this clause). Identifiers of referenced definitions are not automatically imported.

In particular, if module C imports global value or template definitions (e.g. constants, module parameters, templates) or local definitions (e.g. formal parameters of templates, functions, etc., or constants and variables of component types) of an enumerated type from module B, the enumerated values of this type (i.e. the identifiers) are implicitly and automatically imported to module C. That is, the names of the enumerations are known when the enumerated value or template is used in module C (e.g. when an actual parameter is passed or a value is assigned to a component variable). Note that this implicit importing does not impose the restriction given in clause 6.2.4 on global names defined in module C.

e) If the referenced definitions are wished to be used in the importing module, they shall be explicitly imported either directly from its source module or indirectly by importing the import statements of a module importing it (see clause 8.2.3.7).

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

g) The language specification of the import statement shall not override the language specification of the importing module.

h) The language specification of the import statement shall be identical to the language specification of the source module from which definitions are imported (see clause 8.2.3.8) provided a language specification is defined in the source module. If not, the language specification in the import statement is taken as the language specification of the source module. If the source module uses however language concepts not being part of that language specification, this causes an error for the import statement.

Examples

EXAMPLE 1: Selected import examples

```
module MyModuleA
{   :
    // Scope of the imported definitions is global to MyModuleA
    import from MyModuleB all;  // import of all definitions from MyModuleB
    import from MyModuleC {     // import of selected definitions from MyModuleC
        type MyType1, MyType2;  // import of types MyType1 and MyType2
        template all            // import of all templates
    }
    :
    function MyBehaviourC()
    {
        // import cannot be used here
         :
    }
    :
    control
    {
        // import cannot be used here
        :
    }
}
```

EXAMPLE 2: Use of imported definitions and visibility of definitions referenced by them

```
module ModuleONE {

    modulepar integer ModPar1 := …;

    type record RecordType_T1 {
        integer Field1_T1,
        :
    }

    type record RecordType_T2 {
        RecordType_T1   Field1_T2,
        :
    }

    const integer MyConst := …;

    template RecordType_T2 Template_T2 (RecordType_T1 TempPar_T2):= { // parameterized template
        Field1_T2 := …,
        :
    }

} // end module ModuleONE


module ModuleTWO {

    import from ModuleONE {
        template Template_T2
    }

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

    import from ModuleONE {
        modulepar ModPar2
    }

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

} // end module ModuleTWO


module ModuleTHREE {

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

    type port MyPortType message {
        inout RecordType_T2     // Reference to a type defined in ModuleONE
    }

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

    function MyFunction () return integer {
        return MyConst          // Reference to a module constant of ModuleONE
    }

    testcase MyTestCase (out RecordType_T2 MyPar) runs on MyCompType {

        :
        MyPort.send(Template_T2); // Sending a template defined in ModuleONE
        :

    }

} // end ModuleTHREE
```

```
module ModuleFOUR {

    import from ModuleTHREE {
        testcase MyTestCase
    }

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

} // end ModuleFOUR
```

EXAMPLE 3: Handling of name clashes

```
module MyModuleA  {
     :
    type bitstring MyTypeA;

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

NOTE 7: Definitions with the same name defined in different modules are always assumed to be different, even if the actual definitions in the different modules are identical. For example, importing a type that is already defined locally, even with the same name, would lead to two different types being available in the module.

EXAMPLE 4: Name clash between enumerations and global definitions

```
module A {
  type enumerated MyEnumType {enumX, enumY, enumZ}
  type enumerated MyEnumType2 {enumX, enumY, enumZ}
}

module B {
  import from A all;
  const MyEnumType enumY := enumX; // this is allowed as enumeration names restrict
                                   // global names in module A only (see clause 6.2.4)

  const MyEnumType2 enumX := enumX;// this is the enumX value of the type MyEnumType2

  const integer enumZ := 0;

  modulepar MyEnumType px_MyModulePar1 := enumY
  // the default value of the module parameter will be the value enumY, as the type of
  // px_MyModulePar1 creates the context of MyEnumType and in this context enumeration names
  // takes precedence over global definition names; note that for the same context reason there
  // in no name clash between the enumeration names defined in MyEnumType and in MyEnumType2

  modulepar MyEnumType px_MyModulePar2 := B.enumY
  // the default value of the module parameter will be the value enumX, as the prefix
  // identifies the constant definition enumY unambiguously, which has the value enumX

  modulepar integer px_IntegerPar := enumZ;
  // the default value of the module parameter will be 0 as this assignment is not in the
  // context of an enumerated type, hence no name clash occurs

  modulepar MyEnumType px_MyModulePar3 := B.enumX
  // causes an error as px_MyModulePar3 and the constant enumX has different types
}
```

EXAMPLE 5: Importing local definitions transitively

```
module A {
  type enumerated MyEnum_Type { enumX, enumY, enumZ}
  type record MyRec { integer a, integer b }
  type component MyComp { var MyRec v_Rec := { a := 5 } }
}

module B {
  import from A all;
  modulepar MyEnum_Type px_MyModulePar := enumY;
  type component MyCompUser extends MyComp {}
}

module C {
  import from B all;
  testcase TC() runs on MyCompUser {
    if (px_MyModulePar == enumY) {
      // the name enumY is know in C without explicitly importing it from A
      setverdict(pass)
    }
    if (v_Rec.a == 5) {
      v_Rec.b := v_Rec.a;
      // Both the variable name v_Rec and the record field names are known in C without
      // explicitly importing them from A
      setverdict (pass)
    }
  }
}
```

8.2.3.2 Importing single definitions

Single visible definitions can be imported by referring to the definition kind and the definition name(s). The import of single definitions can be used in combination with imports of groups (see clause 8.2.3.3), with imports of definitions of the same kind (see clause 8.2.3.4), and with imports of import statements (see clause 8.2.3.7).

Syntactical Structure

```
[ Visibility ] import from ModuleId "{"
     {
         (
             ( type      { TypeDefIdentifier     [ "," ] } ) |
             ( template  { TemplateIdentifier    [ "," ] } ) |
             ( const     { ConstIdentifier       [ "," ] } ) |
             ( testcase  { TestcaseIdentifier    [ "," ] } ) |
             ( altstep   { AltstepIdentifier     [ "," ] } ) |
             ( function  { FunctionIdentifier    [ "," ] } ) |
             ( signature { SignatureIdentifier   [ "," ] } ) |
             ( modulepar { ModuleParIdentifier   [ "," ] } )
         )
         [ ";" ]
     }
"}" [ ";" ]
```

Semantic Description

See clause 8.2.3. Import of an invisible definition shall cause an error.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The definition to be imported shall be defined in the module from which it is to be imported and shall be visible to the importing module.

b) See the restrictions given in clause 8.2.3.

Examples

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

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

8.2.3.3 Importing groups

Groups of definitions may be imported. The import of groups can be used in combination with imports of single definitions (see clause 8.2.3.2), with imports of definitions of the same kind (see clause 8.2.3.4), and with imports of import statements (see clause 8.2.3.7).

It is allowed to import sub-groups (i.e. a group which is defined within another group) directly, i.e. without the groups in which the sub-group is embedded. If the name of a sub-group that should be imported is identical to the name of another sub-group in the same module (see clause 8.2.2), the dot notation shall be used to identify the sub-group to be imported uniquely.

If some visible definitions of a group are wished not to be imported, their kinds and identifiers shall be listed in the exception list within a pair of curly brackets following the **`except`** keyword. The **`all`** keyword is also allowed to be used in the exception list; this will exclude all definitions of the same kind from the import statement.

Syntactical Structure

```
[ Visibility ] import from ModuleId "{"
    {
        ( group { FullGroupIdentifier [ except "{" ExceptSpec "}" ] [ "," ] } )
        [ ";" ]
    }
"}" [ ";" ]
```

Semantic Description

The effect of importing a group is identical to an **`import`** statement that lists all visible definitions (including sub-groups) of this group except of those that are listed in the except specification. See also clause 8.2.3. Import statements contained in the group or in its subgroups are not part of this list, only definitions are.

It is important to point out, that the except statement does not exclude the definitions listed from being imported in general; all statements importing definitions of the same kind can be seen as a shorthand notation for an equivalent list of identifiers of single definitions. The **`except`** statement excludes definitions from this single list only.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The group to be imported shall be defined in the module from which it is to be imported.

b) See the restrictions given in clause 8.2.3.

Examples

```
import from MyModule { group MyGroup } // includes all visible definitions from MyGroup

import from MyModule {
    group MyGroup except {
        type MyType3, MyType5;  // excludes the two types from the import statement,
        template all            // excludes all templates defined in MyGroup
                                // from the import statement
                                // but imports all other visible definitions of MyGroup
    }
}
```

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

8.2.3.4 Importing definitions of the same kind

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

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

The import of visible definitions of the same kind can be used in combination with imports of single visible definitions (see clause 8.2.3.2), with imports of groups (see clause 8.2.3.3), and with imports of import statements (see clause 8.2.3.7).

Syntactical Structure

```
[ Visibility ] import from ModuleId "{"
    {
        (
            ( type      all [ except { TypeDefIdentifier    [ "," ] } ] ) |
            ( template  all [ except { TemplateIdentifier   [ "," ] } ] ) |
            ( const     all [ except { ConstIdentifier      [ "," ] } ] ) |
            ( testcase  all [ except { TestcaseIdentifier   [ "," ] } ] ) |
            ( altstep   all [ except { AltstepIdentifier    [ "," ] } ] ) |
            ( function  all [ except { FunctionIdentifier   [ "," ] } ] ) |
            ( signature all [ except { SignatureIdentifier  [ "," ] } ] ) |
            ( modulepar all [ except { ModuleParIdentifier  [ "," ] } ] )
        )
        [ ";" ]
    }
"}" [ ";" ]
```

Semantic Description

The effect of importing definitions of the same kind is identical to an **import** statement that lists all visible definitions of that kind except of those that are listed in the except specification. See also clause 8.2.3.

NOTE: If the list of all visible definitions of that kind except of those that are listed in the **except** specification is empty, the import statement has no effect. This case does not lead to an error.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) See the restrictions given in clause 8.2.3.

Examples

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

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

8.2.3.5 Importing all definitions of a module

All visible definitions of a module definitions part may be imported using the **all** keyword next to the module name.

If some visible definitions are wished not to be imported, their kinds and identifiers shall be listed in the exception list within a pair of curly brackets following the **except** keyword. The **all** keyword is also allowed to be used in the exception list; this will exclude all visible declarations of the same kind from the import statement.

NOTE 1: If the list of all visible definitions of a module except of those that are listed in the **except** specification is empty, the import statement has no effect. This case does not lead to an error.

NOTE 2: Importing all definitions of a module imports only definitions declared directly in that module, but does not import the import statements of that module (see also clause 8.2.3.7).

Syntactical Structure

```
[ Visibility ] import from ModuleId
     all
     [
         {
             except "{"
                 ( group     { FullGroupIdentifier   [ "," ] } | all ) |
                 ( type      { TypeDefIdentifier     [ "," ] } | all ) |
                 ( template  { TemplateIdentifier    [ "," ] } | all ) |
                 ( const     { ConstIdentifier       [ "," ] } | all ) |
                 ( testcase  { TestcaseIdentifier    [ "," ] } | all ) |
                 ( altstep   { AltstepIdentifier     [ "," ] } | all ) |
                 ( function  { FunctionIdentifier    [ "," ] } | all ) |
                 ( signature { SignatureIdentifier   [ "," ] } | all ) |
                 ( modulepar { ModuleParIdentifier   [ "," ] } | all )
             "}"
             [ ";" ]
         }
     ]
[ ";" ]
```

Semantic Description

The effect of importing all visible definitions of a module is identical to an **import** statement that lists all importable definitions of that module except of those that are listed in the except specification. See also clause 8.2.3.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) If all visible definitions of a module are imported by using the all keyword, no other form of import (import of single definitions, import of the same kind, etc.) shall be used for the same import statement.

b) In the set of except statements for an all import, only one except statement per kind of definition (i.e. for a group, type, etc.) is allowed.

Examples

```
import from MyModule all;   // includes all definitions from MyModule

import from MyModule all except {
    type MyType3, MyType5;  // excludes these two types from the import statement and
    template all            // excludes all templates declared in MyModule,
                            // from the import statement
                            // but imports all other definitions of MyModule
}
```

8.2.3.6 Import definitions from other TTCN-3 editions and from non-TTCN-3 modules

In cases when visible definitions are imported from modules from other TTCN-3 editions or from other sources than TTCN-3 modules, the language specification shall be used to denote the language (may be together with a version number) of the source (e.g. module, package, library or even file) from which definitions are imported. It consists of the **language** keyword and a subsequent textual declaration of the denoted language.

The use of the language specification is optional when importing from a TTCN-3 module of the same edition as the importing module. The TTCN-3 language identifiers defined in clause 8.1 are to be used. Package identifiers from ES 202 781 [i.11], ES 202 782 [i.14], ES 202 784 [i.12] and ES 202 785 [i.13] can be used in addition. Identifiers for other languages are defined in the language mapping parts of TTCN-3, i.e. in ES 201 873-7 [i.5], ES 201 873-8 [i.6] and ES 201 873-9 [i.7].

When an incompatibility is discovered between the language and/or package identification (including implicit identification by omitting the language specification) and the syntax of the module from which definitions are imported, tools shall provide reasonable effort to resolve the conflict.

Syntactical Structure

```
[ Visibility ] import from ModuleIdentifier [ LanguageSpec ] … [ ";" ]
```

Semantic Description

TTCN-3 supports the referencing of elements defined in other TTCN-3 editions (*versioned elements*) or other languages (*foreign elements*) from within TTCN-3 modules. Such elements can be used in a TTCN-3 module of a given edition only if they have a TTCN-3 view in that TTCN-3 edition. The term TTCN-3 view can be best explained by considering the case when the definition of a TTCN-3 element is based on another TTCN-3 element, the information content of the referenced element shall be available and is used for the new definition. For example, when a template is defined based on a structured type, the identifiers and types of fields of the base type shall be accessible and are used for the template definition. In a similar way, when a base type is a versioned or foreign element it shall provide the same information content as would be required from a TTCN-3 type declaration. The versioned or foreign element, naturally, may contain more information than required by TTCN-3. The TTCN-3 view of a versioned or foreign element means that part of the information carried by that element, which is necessary to use it in TTCN-3. Obviously, the TTCN-3 view of a versioned or foreign element may be the full set or a subset of the information content of that element but never a superset. There may be versioned or foreign element without a TTCN-3 view (zero TTCN-3 view), i.e. for some reason no TTCN-3 definition in the given edition could be based on them.

To make declarations of versioned or foreign element visible in TTCN-3 modules, their names shall be imported just like definitions in other TTCN-3 modules of the given edition. When imported, only the TTCN-3 view of the versioned or foreign element will be seen from the importing TTCN-3 module. There are two main differences between importing TTCN-3 elements of the same editions and versioned or foreign elements:

- to import from a TTCN-3 module of another edition of from a non-TTCN-3 module the import statement shall contain an appropriate language identifier string;
- only versioned or foreign elements with a TTCN-3 view of a given edition are importable into a TTCN-3 module of that edition.

Importing can be done automatically using the all directive, in which case all importable objects shall automatically be selected by the testing tool, or done manually by listing names of elements to be imported. Naturally, in the second case only importable elements are allowed in the list.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The language specification may only be omitted if the referenced module contains TTCN-3 notation and the TTCN-3 version is known.

b) Definitions imported from non-TTCN-3 language sources have by default public visibility provided that no other rules are defined in the respective language mapping (see ES 201 873-7 [i.5], ES 201 873-8 [i.6] or ES 201 873-9 [i.7], respectively).

Examples

```
module MyNewModule {
  import from MyOldModule language "TTCN-3:2003" {
    type MyType
  }
}
```

```
module MyNewestModule {
  import from MyNewModule language "TTCN-3:2010" { import all };
  // the language specifications shall be identical, see clause 8.2.3.8
}
```

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

8.2.3.7 Importing of import statements from TTCN-3 modules

Visible import statements of TTCN-3 modules can be imported by other TTCN-3 modules.

Syntactical Structure

```
[ Visibility ] import from ModuleIdentifier [ LanguageSpec ]
               "{" import all [ ";" ] "}" [ ";" ]
```

Semantic Description

TTCN-3 supports importing of visible import statements from other TTCN-3 modules. This means that import statements of the module, from which the import statements are imported, are re-imported to the importing module. For example, if module B imports the import statements of module A, everything that is imported by A using import statements visible for module B, is also imported by B. If another module C imports all import statements from B, then C imports all what A is importing - provided that the import statements are visible to modules B and C.

It is not possible to import individual import statements of another module.

The import of import statements can be used in combination with imports of single definitions (see clause 8.2.3.2), with imports of groups (see clause 8.2.3.3), and with imports of definitions of the same kind (see clause 8.2.3.4).

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The restrictions given in clause 8.2.3.1 apply.

b) The restrictions given in clause 8.2.3.6 apply.

c) Importing of import statements is only possible from other TTCN-3 modules, i.e. the language specification shall denote a TTCN-3 edition only, not a non-TTCN-3 language.

Examples

EXAMPLE: Importing of visible import statements

```
module A {
    type integer T1;
    type integer T2;
    template T1 t1 := ?;
    template T2 t2 := *;
    :
}
module B {
    public import from A { type T1 }
    type charstring T2;
    template T1 t1 := ( 1, 2, 3 );
    :
}
module C {
    public import from B { import all }  // imports the import statements only
    public import from B { type T2 }     // imports the type B.T2
    import from A { template all }
    :
}
module D {
    private import from C { import all } // imports the import statements only
    :
}
module E {
    import from D { import all }
```

```
    :
}

// yields the following
// module A knows
// A.T1     (defined)
// A.T2     (defined)
// A.t1     (defined)
// A.t2     (defined)
//
// module B knows
// A.T1     (imported)
// B.T2     (defined)
// B.t1     (defined)
//
// module C knows
// A.T1     (imported from B importing it from A)
// B.T2     (imported)
// A.t1     (imported)
// A.t2     (imported)
//
// module D knows
// A.T1     (imported from C importing it from B importing it from A)
// B.T2     (imported from C importing it from B)
// A.t2 and A.t2 are not imported as their imports are private to C
//
// module E "knows" nothing
// as the imports of D are private and not visible to E
```

8.2.3.8 Compatibility of language specifications in imports

When importing into a TTCN-3 module, the language specification of the importing module, the language specification of the import statement and the language specification of the source module, where the imported definitions are defined, have to be compatible according to the following rules.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) A TTCN-3 module of any TTCN-3 edition can import from a non-TTCN-3 language source provided that a TTCN-3 view for the non-TTCN-3 language exists (see clause 8.2.3.6).

b) Definitions or import statements are imported according to the language specification in which the definition or the import statement is defined. If no language specification is given in this module, the language specification of the import statement with which those definitions or import statements are to be imported, is used instead. If the module, within which the definitions or the import statements are defined, and the import statement for these definitions or import statements provide both a language specification, then they shall be identical. If none of the two has a language specification, the language specification has to be known from other sources, which is tool specific.

c) The TTCN-3 language specification in an import statement shall be lower or equal to the TTCN-3 language specification of the importing module, i.e. a TTCN-3 module can only import from earlier or same editions of TTCN-3 but not from later editions.

8.2.4 Definition of friend modules

Modules can define other modules to be friends.

Syntactical Structure

```
[ private ] friend module ModuleIdentifier { "," ModuleIdentifier } ";"
```

Semantic Description

Friendship to modules is defined by the exporting module (the module that declares the definitions) not by the importing module (the module that uses the module definitions of another module). Friendship can be cyclic.

If a module is friend to a module from which it imports top-level definitions, all top-level definitions with public and friend visibility are visible to the friend module. For non-friend modules, public top-level definitions are visible only.

Missing friend modules shall not cause an error.

NOTE: Friend modules can be checked by tools, however at most warning are to be issued if a friend module is missing.

Restrictions

In addition to the general static rules of TTCN 3 given in clause 5, the following restrictions apply:

a) Only private visibility can be defined for friend definitions as they are always private.

Examples

```
module MyModuleA {
    friend module MyModuleB,MyModuleC;
}
// MyModuleB and MyModuleC are friends of MyModuleA

module MyModuleB {
    friend module MyModuleA;
}
// MyModuleA is friend of MyModuleB

module MyModuleC {
}
```

8.2.5 Visibility of definitions

Top-level module definitions and import statements have a visibility, which can be explicitly set. They are by default **public** except for imported and friend definitions. Import definitions are by default **private**. Friend definitions are **private** only. Group definitions are **public** only.

Syntactical Structure

```
[ public | friend | private ]
```

Semantic Description

The visibility controls whether a top-level definition or an import statement is importable by another module.

Three visibilities are distinguished:

- A top-level definition or an import statement with **public** visibility is importable by any other module.
- A top-level definition or an import statement with **friend** visibility is importable by friend modules only (see clause 8.2.4).
- A top-level definition or an import statement with **private** visibility cannot be imported at all.

NOTE: As specified in restriction e) of clause 8.2.3.1, this means that importable definitions are imported together with all information of referenced definitions that are necessary for the usage of the importable definition, even if the referenced definition is private. Only the identifier of the referenced definition is not visible in the importing TTCN-3 module.

The visibility of groups is always **public**. The visibility of imported definitions is by default **private**. All other module definitions are by default **public**.

The visibility of a top-level definition or an import statement defines their importability by another module. If the top-level definition or the import statement is part of a group, this has no effect on the importability of the module definition. The importability of a top-level definition by another module is summarized in table 9, the importability of import statements in table 10.

Table 9: Visibility and import of module definitions

Visibility of module definition	Module definition importable directly by a non-friend module	Module definition importable directly by a friend module	Module definition importable via group import by a non-friend module	Module definition importable via group import by a friend module
public	yes	yes	yes	yes
friend	no	yes	no	yes
private	no	no	no	no

Table 10: Visibility and import of import statements

Visibility of import	Import imported by a non-friend module	Import imported by a friend module
public	yes	yes
friend	no	yes
private	no	no

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

```
module MyModuleA {
    friend module MyModuleC;
    private type integer MyInteger;
    // MyInteger is not visible to other modules
    friend type charstring MyString;
    // MyString is visible to friend modules
    public type boolean MyBoolean;
    // MyBoolean is visible to all modules
}
module MyModuleB {
    import from MyModuleA all;
    // MyString and MyInteger are not visible and are not imported
    // MyBoolean is imported
}
module MyModuleC {
    import from MyModuleA all;
    // MyInteger is not visible and is not imported
    // MyString and MyBoolean are imported
}
```

8.3 Module control part

The module control part may contain local definitions (i.e. constants or templates), local instances (i.e. variables or timers) and describe the selection, parameterization and execution order (possibly repetitive) of the actual test cases. A test case shall be defined in the module definitions part or imported from another module, and called in the control part.

The control part of a module calls the test cases with actual parameters and controls their execution. Program statements can be used to specify the selection and execution order of the test cases. Definitions made in the module control part have local visibility, i.e. can be used within the control part only.

This is explained in more detail in clause 26.

EXAMPLE:

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

9 Port types, component types and test configurations

TTCN-3 allows the (dynamic) specification of concurrent test configurations (or configuration for short). A configuration consists of a set of inter-connected test components with well-defined communication ports and an explicit test system interface which defines the borders of the test system (see figure 4).

NOTE: Additional configuration and deployment support for TTCN-3 is defined in the optional package [i.11].

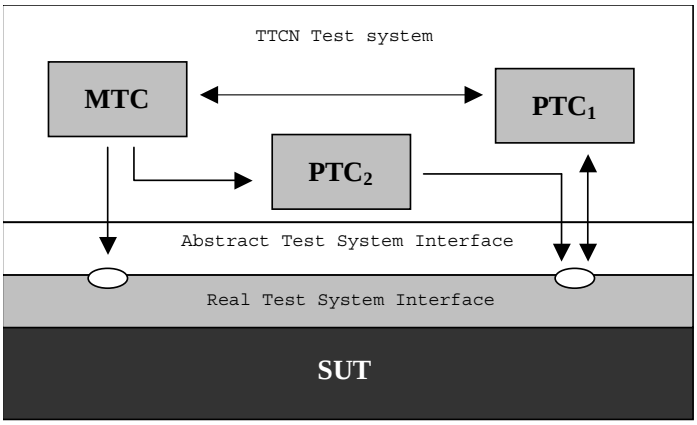


Figure 4: Conceptual view of a typical TTCN-3 test configuration

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

Test case execution shall end when the MTC terminates. All other PTCs are treated equally i.e. there is no explicit hierarchical relationship among them and the termination of a single PTC terminates neither other components nor the MTC. When the MTC terminates, the test system has to stop all PTCs not terminated by the moment when the test case execution is ended.

Communication between test components and between the components and the test system interface is achieved via communication ports (see clause 9.1).

Test component types and port types, denoted by the keywords **component** and **port**, shall be defined in the module definitions part. The actual configuration of components and the connections between them is achieved by performing **create** and **connect** operations within the test case behaviour. The component ports are connected to the ports of the test system interface by means of the **map** operation (see clause 21.1.1).

9.1 Communication ports

Test components are connected via their ports, i.e. connections among components and between a component and the test system interface are port-oriented. Each port is modelled as an infinite FIFO queue which stores the incoming messages or procedure calls until they are processed by the component owning that port (see figure 5).

NOTE: While TTCN-3 ports are infinite in principle in a real test system they may overflow. This is to be treated as a test case error (see clause 24.1).

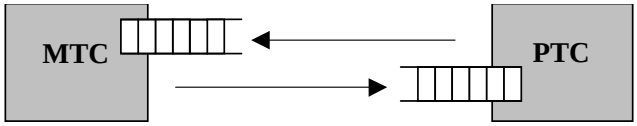


Figure 5: The TTCN-3 communication port model

TTCN-3 connections are port-to-port and port-to-test system interface connections (see figure 6). There are no restrictions on the number of connections a component may maintain. One-to-many connections are also allowed (e.g. figure 6(g) or figure 6(h)).

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The following connections are not allowed (see figure 7):

- A port owned by a component A shall not be connected with two or more ports owned by the same component (figures 7 (a) and 7(e)).
- A port owned by a component A shall not be connected with two or more ports owned by a component B (see figure 7(c)).
- A port owned by a component A can only have a one-to-one connection with the test system interface. This means, connections as shown in figures 7(b) and 7(d) are not allowed.
- Connections within the test system interface are not allowed (see figure 7(f)).
- A port that is connected shall not be mapped and a port that is mapped shall not be connected (see figure 7(g)).

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

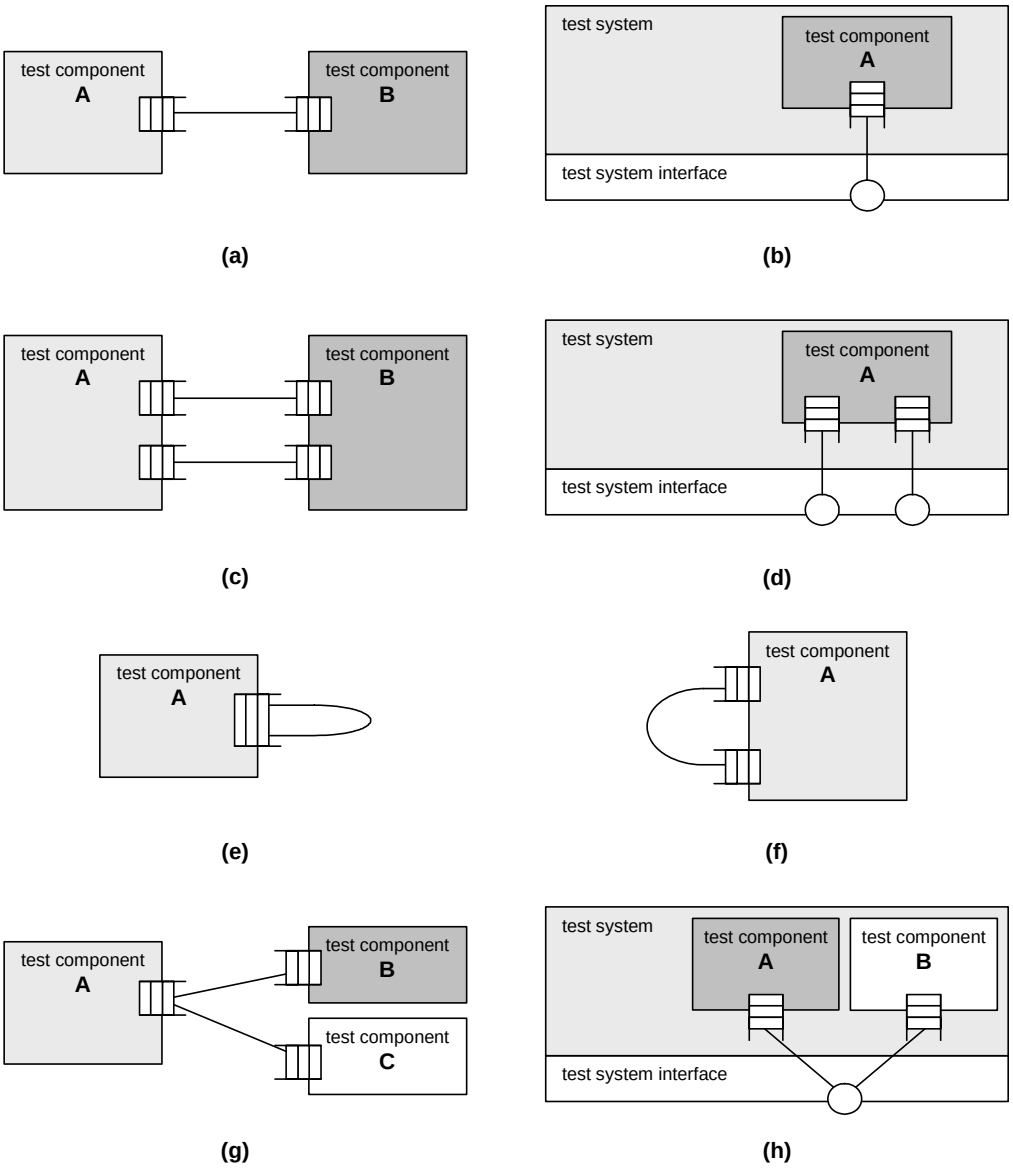


Figure 6: Allowed connections

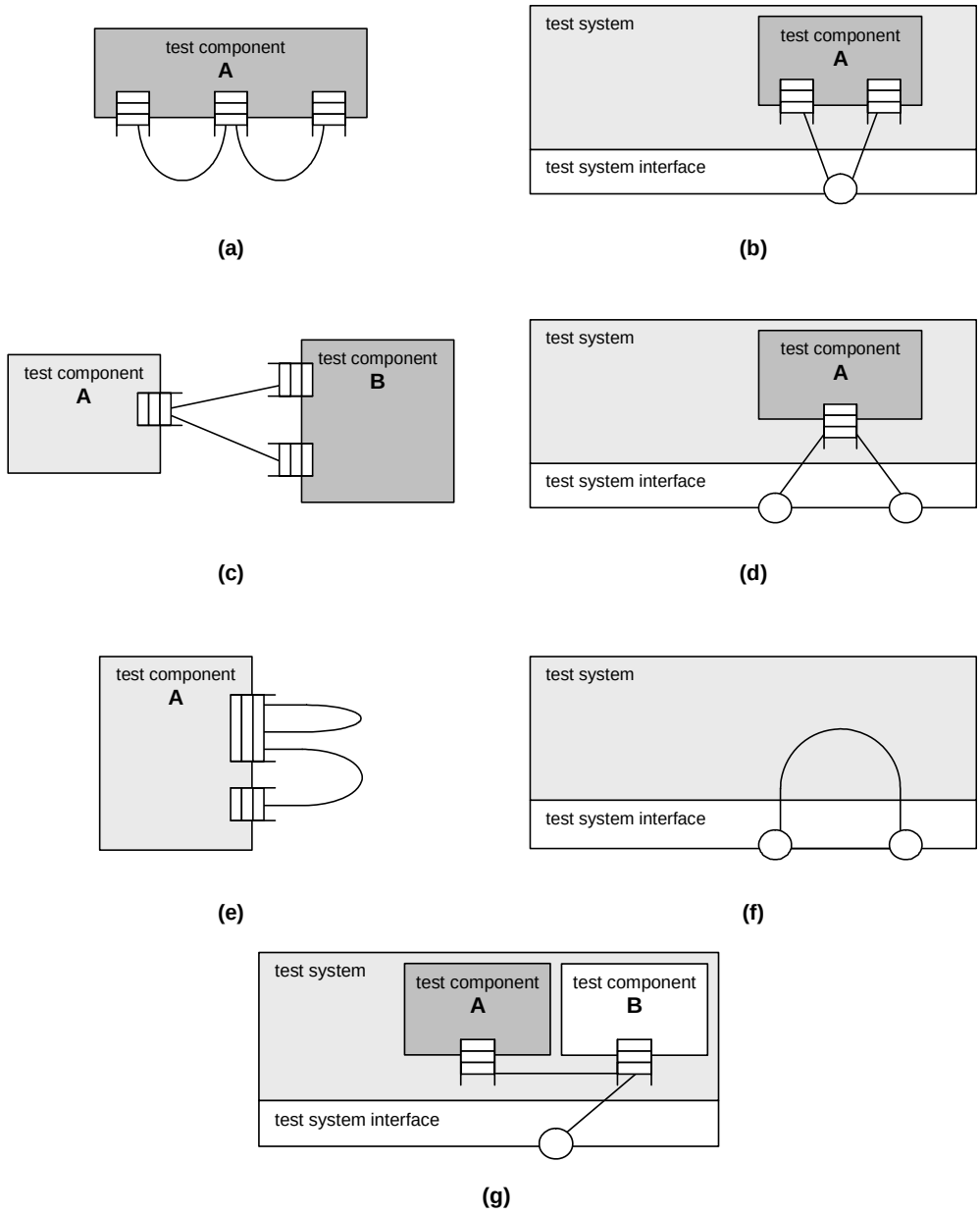


Figure 7: NOT allowed connections

9.2 Test system interface

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

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

The test system interface statically defines the number and type of the port connections to the SUT during a test run. However, the connections between the test system interface and the TTCN-3 test components are dynamic in nature and may be modified during a test run by using **`map`** and **`unmap`** operations (see clause 21.1).

A component type definition is used to define the test system interface because, conceptually, component type definitions and test system interface definitions have the same form (both are collections of ports defining possible connection points). When used as test system interfaces, components cannot make use of any constants, variables and timers declared in the component type.

Syntactical Structure

The same as a component type definition (see clauses 6.2.11 and 6.2.11.2).

Semantic Description

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

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

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

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

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The same as for component type definitions (see clauses 6.2.11 and 6.2.11.2).

Examples

EXAMPLE 1: Explicit definition of a test system interface

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

type component MyTestSystemInterface
{
    port MyMessagePortType      PCO1, PCO2;
    port MyProcedurePortType    PCO3
}

// MyTestSystemInterface is the test system interface
testcase MyTestcase1 () runs on MyMTCType system MyTestSystemInterface {
    // establishing the port connections
    map(mtc:PCO1, system:PCO2);
    // the testcase behaviour
    // …
}
```

EXAMPLE 2: Implicit definition of a test system interface

```
// MyMTCType is the test system interface
testcase MyTestcase2 () runs on MyMTCType {
    // map statements are not needed
    // the testcase behaviour
    // …
}
```

10 Declaring constants

TTCN-3 constants are run-time constants. After value assignment, they do not change their value during test execution. They can be used on the right hand side of assignments, in expressions, in actual parameters, and in template definitions. Constants used within type definitions have to have values known at compile-time.

Syntactical Structure

```
const Type { ConstIdentifier [ ArrayDef ] ":=" ConstantExpression [ "," ] } [ ";" ]
```

Semantic Description

A constant assigns a name to a fixed value. A value is assigned only once to a constant, at the place of its declaration. The constant does not change its value during test execution. The constant is defined only once, but can be referenced multiple times in a TTCN-3 module.

Optional fields of record and set constants or constant fields can be initialized explicitly or implicitly. For implicit initialization of the optional fields of a constant or a constant field, an **optional** attribute with the value **"implicit omit"** (see clause 27.7) shall be associated with it either directly or via the attribute distribution (scoping) mechanism (see clause 27.1.1).

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) Constants shall not be of port type.

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

b) Constant expressions initializing constants, which are used in type and array definitions, shall only contain literals, predefined functions except of `rnd` (see clause 16.1.2), operators specified in clause 7.1, and other constants obeying the limitations of this paragraph.

Examples

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

11 Declaring variables

TTCN-3 variables are statically typed variables. Variables are either value variables to store values or template variables to store templates.

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

Variables can be declared and used in the module control part, test cases, functions and altsteps. Additionally, variables can be declared in component type definitions. These variables can be used in test cases, altsteps and functions which are running on a given component type.

11.1 Value variables

A TTCN-3 value variable stores values. It is declared by the **var** keyword followed by a type identifier and a variable identifier. An initial value can be assigned at variable declaration.

It may be used at the right hand side as well as at the left hand side of assignments, in expressions, following the **return** keyword in bodies of functions with a return clause in their headers and may be passed to both value and template-type formal parameters.

Syntactical Structure

```
var Type VarIdentifier [ ArrayDef ] [ ":=" Expression ]
        { [ "," ] VarIdentifier [ ArrayDef ] [ ":=" Expression ] } [ ";" ]
```

Semantic Description

A value variable associates a name with the location of a value. A value variable may change its value during test execution several times. A value can be assigned several times to a value variable. The value variable can be referenced multiple times in a TTCN-3 module.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) *Expression* shall be of type *Type*.

b) Value variables shall store values only.

c) Value variables shall not be declared or used in a module definitions part (i.e. global variables are not supported in TTCN-3).

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

Examples

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

11.2 Template variables

A TTCN-3 template variable stores templates. They are declared by the **var template** keyword followed by a type identifier and a variable identifier. An initial content can be assigned at declaration. In addition to values, template variables may also store matching mechanisms (see clause 15.7).

Template variables may be used on the right hand side as well as on the left hand side of assignments, following the **return** keyword in bodies of functions defining a template-type return value in their headers and may be passed as actual parameters to template-type formal parameters. It is also allowed to assign a template instance to a template variable or a template variable field.

Syntactical Structure

```
var template [ restriction ] Type VarIdentifier [ ArrayDef ] ":=" TemplateBody
                { [ "," ] VarIdentifier [ ArrayDef ] ":=" TemplateBody } [ ";" ]
```

Semantic Description

A template variable associates a name with the location of a template or a value (as every value is also a template). A template variable may change its template during test execution several times. A template or value can be assigned several times to a template variable. The template variable can be referenced multiple times in a TTCN-3 module.

The content of a template variable can be restricted to the matching mechanisms specific value and omit in the same way as formal template parameters, see clause 5.4.1.2. The restriction **template (omit)** can be replaced by the shorthand notation **omit**.

NOTE 1: String and list type templates can be concatenated, see clause 15.11.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) Template variables shall not be declared or used in a module definitions part (i.e. global variables are not supported in TTCN-3).

b) When used on the right hand side of assignments template variables shall not be operands of TTCN-3 operators (see clause 7.1) and the variable on the left hand side shall be a template variable too.

c) When accessing element of template variables either on the left hand side or on the right hand side of assignments, the rules given in clause 15.6 shall apply.

NOTE 2: While it is not allowed to directly apply TTCN-3 operations to template variables, it is allowed to use the dot notation and the index notation to inspect and modify template variable fields.

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

e) If the template variable is restricted, then the template used to initialize it shall contain only the matching mechanisms as described in clause 15.8.

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

g) Restrictions on templates in clause 15 shall apply.

Examples

```
var template integer  MyVarTemp1 := ?;
var template MyRecord MyVarTemp2 := { field1 := true, field2 := * },
                      MyVarTemp3 := { field1 := ?, field2 := MyVarTemp1 };
```

12 Declaring timers

TTCN-3 provides a timer mechanism. Timers can be declared and used in the module control part, test cases, functions and altsteps. Additionally, timers can be declared in component type definitions. These timers can be used in test cases, functions and altsteps which are running on the given component type.

A timer declaration may have an optional default duration value assigned to it. The timer shall be started with this value if no other value is specified. The timer value shall be a non-negative **`float`** value (i.e. greater than or equal to 0.0) where the base unit is seconds.

In addition to single timer instances, timer arrays can also be declared. Default duration(s) of the elements of a timer array shall be assigned using a value array. Default duration(s) assignment shall use the array value notation as specified in clause 6.2.7. If the default duration assignment is wished to be skipped for some element(s) of the timer array, it shall explicitly be declared by using the not used symbol ("-").

Syntactical Structure

```
timer { TimerIdentifier [ ArrayDef ] ":=" TimerValue [ "," ] } [ ";" ]
```

Semantic Description

Timers are local to components. A component can start and stop a timer, check if a timer is running, read the elapsed time of a running timer and process timeout events after timer expiration. The timer value is interpreted with a base unit of seconds.

NOTE 1: Timers declared and started in scope units such as functions cease to exist when the scope unit is left. They do not contribute to the test behaviour once the scope unit is left.

NOTE 2: It is not possible to define a timer array as type.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) In case of a single timer, the default duration value shall resolve to a non-negative numerical float value (i.e. the value shall be greater or equal 0.0, infinity and not_a_number are disallowed).

b) In case of a timer array, it shall resolve to an array of float values obeying to restriction a) above of the same size as the size of the timer array.

Examples

EXAMPLE 1: Single timer

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

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

EXAMPLE 2: Timer array

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

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

13 Declaring messages

One of the key elements of TTCN-3 is the ability to send and receive simple or complex messages over message-based ports defined by the test configuration (see clauses 9 and 21). These messages may be those explicitly concerned with testing the SUT or with the internal co-ordination and control messages specific to the relevant test configuration.

Messages are instances of types declared in the in/out/inout clauses of message port type definition.

Any type can be declared as type of a message in a message port type definition, i.e. values of any basic or structured type (see clauses 6.1 and 6.2) can be sent or received. Received messages can also be declared as a combination of value and matching mechanisms (see clause 15.5). Instances of messages can be declared by global, local or in-line templates (see clause 15) or being constructed and passed via variables or template variables (see clause 11) and parameters or template parameters (see clause 5.4).

Syntactical Structure

See syntactical structure of types (see clause 6).

Semantic Description

See semantic description of types (see clause 6).

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

```
// a structured, ordered message with two fields
type record ARecord { integer i, float f }
```

14 Declaring procedure signatures

Procedure signatures (or signatures for short) are needed for procedure-based communication. Procedure-based communication may be used for the communication within the test system, i.e. among test components, or for the communication between the test system and the SUT. In the latter case, a procedure may either be invoked in the SUT (i.e. the test system performs the call) or in the test system (i.e. the SUT performs the call).

Syntactical Structure

```
signature SignatureIdentifier
"(" { [ in | inout | out ] Type ValueParIdentifier [ ","] } ")"
[ ( return Type ) | noblock ]
[ exception "(" ExceptionTypeList ")" ]
```

Semantic Description

For all used procedures, i.e. procedures used for the communication among test components, procedures called from the SUT and procedures called from the test system, a procedure **signature** shall be defined in the TTCN-3 module.

TTCN-3 supports *blocking* and *non-blocking* procedure-based communication. By default, signature definitions without the **noblock** keyword are assumed to be used for blocking procedure-based communication.

Signature definitions may have parameters. Parameters shall be of data type only, i.e. of a basic type, a structured type thereof or a subtype thereof. Within a **signature** definition the parameter list may include parameter identifiers, parameter types and their direction, i.e. **in**, **out**, or **inout**. The direction **inout** and **out** indicate that these parameters are used to retrieve information from the remote procedure.

NOTE 1: The direction of the parameters is as seen by the *called* party rather than the *calling* party.

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

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

NOTE 2: The conversion of exceptions generated by or sent to the SUT into the corresponding TTCN-3 type or SUT representation is tool and system specific and therefore beyond the scope of the present document.

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

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) Signature definitions for non-blocking communication shall use the **noblock** keyword, shall only have **in** parameters and shall have no return value but may raise exceptions.

b) Signature parameters shall not be of port, component or default type or of structured types having fields of port, component or default type.

Examples

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

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

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

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

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

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

15 Declaring templates

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

Templates provide the following possibilities:

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

b) they can be parameterized;

c) they allow matching mechanisms;

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

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

A modified template declaration (see clause 15.5) specifies only the fields to be changed from the base template, i.e. it is a partial specification.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) Templates shall not be of **default** type.

b) Structured type or signature templates shall not include a field of **`default`** type, neither directly, nor by nesting or referencing a structured type or signature that contains a default field.

NOTE: The **`anytype`** type does not include the **`default`** type (see clause 6.2.6), so that restriction b) does not apply to anytype templates.

Examples

```
type record MyRecord {
  default def
}
type union MyUnion {
  integer choice1,
  MyRecord choice2
}
template MyUnion t_integerChosen := { choice1 := 5 }
  // shall cause an error as the type MyUnion contains MyRecord, which includes
  // a field of default type.
```

15.1 Declaring message templates

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

Syntactical Structure

See syntactical structure of global and local templates (see clause 15.3) and of in-line templates (see clause 15.4).

Semantic Description

A template used in a **`send`** operation defines a complete set of field values comprising the message to be transmitted over a port.

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

A template used in a **`receive`**, **`trigger`** or **`check`** operation defines a data template against which an incoming message is to be matched. Matching mechanisms, as defined in clauses 15.7 and 15.8 and in annex B, may be used in receive templates. No binding of the incoming values to the template shall occur.

Restrictions

In addition to restrictions in clause 15, the following restrictions apply:

a) At the time of a **`send`** operation, the used template shall be completely initialized and all fields shall resolve to actual values or to omit and no other matching mechanisms shall be used in the template fields, neither directly nor indirectly.

At the time of a **`receiving`** operation, the matching template shall be completely initialized.

b) Optional fields of record and set templates or template fields can be initialized explicitly or implicitly. For implicit initialization of the optional fields of a template or a template field, an **`optional`** attribute with the value **`"implicit omit"`** (see clause 27.7) shall be associated with it either directly or via the attribute distribution (scoping) mechanism (see clause 27.1.1).

Examples

EXAMPLE 1: Template for sending messages

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

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

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

EXAMPLE 2: Template for receiving messages

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

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

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

EXAMPLE 3: Template for receiving messages

```
// When used in a receiving operation this template will match any integer value
template integer Mytemplate := ?;

// This template will match only the integer values 1, 2 or 3
template integer Mytemplate := (1, 2, 3);
```

15.2 Declaring signature templates

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

Syntactical Structure

See syntactical structure of global and local templates (see clause 15.3) and of in-line templates (see clause 15.4).

Semantic Description

A signature template defines the values and matching mechanisms of the procedure parameters only, but not the return value. The values or matching mechanisms for a return have to be defined within the reply (see clause 22.3.3) or getreply operation (see clause 22.3.4).

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

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

Restrictions

In addition to restrictions in clause 15, the following restrictions apply:

a) At the time of a **call**, **reply** and **raise** operation, the used template shall be completely initialized and all **in**/**inout** parameters in a **call**, all **out**/**inout** parameters in a **reply** or **raise** operation shall resolve to specific values or to omit and no other matching mechanisms shall be used for these parameters, neither directly nor indirectly.

b) The NotUsedSymbol shall only be used in signature templates for parameters which are not relevant and in modified template declarations and modified in-line templates to indicate no change for the specified field or element.

At the time of a **getcall**, **getreply** and **catch** operation, the matching template shall be completely initialized.

c) Optional fields of record and set parameters or parameter fields can be initialized explicitly or implicitly. For implicit initialization of a parameter or a parameter field, an **optional** attribute with the value **"implicit omit"** (see clause 27.7) shall be associated with it either directly or via the attribute distribution (scoping) mechanism (see clause 27.1.1).

Examples

EXAMPLE 1: Templates for invoking and accepting procedures

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

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

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

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

template RemoteProc Template4:=?;
```

EXAMPLE 2: In-line templates for invoking procedures

```
// Given example 1 in this clause

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

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

// Invalid invocation causing an error
// since the inout parameter Par3 has a matching attribute not a value
MyPCO.call(RemoteProc:Template3);

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

EXAMPLE 3: In-line templates for accepting procedure invocations

```
// Given example 1 in this clause

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

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

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

EXAMPLE 4: In-line templates for accepting procedure replies

```
// Given example 1 in this clause

// Valid getreply, in-parameters will be ignored, matches if return value is 4
MyPCO.getreply(RemoteProc:Template2 value 4);

// Valid getreply, accepting any reply for RemoteProc
MyPCO.getreply(RemoteProc:?);

// Valid getreply, also accepting any reply for RemoteProc
MyPCO.getcall(RemoteProc:Template4 value ?);
```

15.3 Global and local templates

TTCN-3 allows defining global templates and local templates.

Syntactical Structure

```
template [ restriction ] Type TemplateIdentifier  ["(" TemplateFormalParList ")"]
[ modifies TemplateRef ] ":=" TemplateBody
```

NOTE: The optional restriction part is covered by clause 15.8.

Semantic Description

Global templates can be defined in the module definitions part. Local templates can be defined in module control, testcases, functions, altsteps or statement blocks. Both global and local templates scoping rules specified in clause 5 apply.

Both global and local templates can be parameterized. The actual parameters of a template can include values and templates. The rules for formal and actual parameter lists shall be followed as defined in clause 5.2.

At the time of their use (e.g. in communication operations **send**, **receive**, **call**, **getcall,** etc.), it is allowed to change template fields by in-line modified templates, to pass in values via value parameters as well as to pass in templates via template parameters. The effects of these changes on the values of the template fields do not persist in the template subsequent to the corresponding communication event.

Restrictions

In addition to restrictions in clause 15, the following restrictions apply:

a) The dot notation such as *MyTemplateId.FieldId* shall not be used to set or retrieve values in templates in communication events. The "->" symbol shall be used for this purpose (see clause 23).

b) Restrictions on referencing elements of templates or template fields are described in clause 15.6.

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

Examples

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

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

15.4 In-line Templates

Templates can be specified directly at the place they are used. Such templates are called in-line templates.

Syntactical Structure

```
[ Type ":" ] [ modifies TemplateRefWithParList ":=" ] TemplateBody
```

NOTE 1: An in-line template is an argument of a communication operation or an actual parameter of a testcase, function or altstep call, i.e. it is always placed within parenthesis and potentially separated with a comma.

Semantic Description

In-line templates can be defined directly at the place of its use.

In-line templates do not have names, therefore they cannot be referenced or reused. The lifetime of in-line templates is the TTCN-3 statement (an assignment, a testcase/function/alstep invocation, a return from a function, a communication operation), where they are defined.

Restrictions

In addition to restrictions in clause 15, the following restrictions apply:

a) Templates may be specified for any TTCN-3 type defined in table 3 and for any procedure signature except for **port** and **default** types.

b) The type field may only be omitted when the type is implicitly unambiguous.

NOTE 2: For literal in-line templates, the following types may be omitted: **integer**, **float**, **boolean**, **bitstring**, **hexstring**, **octetstring**.

NOTE 3: Types of constants, parameters and variables of the actual scope are always unambiguous and can hence always be omitted.

c) In-line templates containing instead of values or inside values matching mechanisms (see clause 15.7) can only be defined in arguments of receiving communication operations (i.e. **receive, trigger, check, getcall**, **getreply** and **catch**), in arguments of the **match** and **select case** operations, in actual template parameters, at the right hand side of assignments (when there is a template variable at the left hand side of the assignment) and in return statements of template returning functions. In-line templates not containing matching mechanisms can be defined wherever values are allowed.

d) When used in communication operations, the type of the in-line template shall be in the port list over which the template is sent or received. In the case where there is an ambiguity between the listed type and the type of the value provided (e.g. through subtyping) then the type name of the in-line template shall be included in the communication operation.

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

Examples

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

15.5 Modified templates

Normally, a template specifies a set of base or default values or matching symbols for each and every field defined in the appropriate type or signature definition. In cases where small changes are needed to specify a new template, it is possible to specify a modified template. A modified template specifies modifications to particular fields of the original template, either directly or indirectly. As well as creating explicitly named modified templates, TTCN-3 allows the definition of in-line modified templates.

Syntactical Structure

Global or local modified template:

```
template [restriction] Type TemplateIdentifier  ["(" TemplateFormalParList ")"]
modifies TemplateRef ":=" TemplateBody
```

NOTE: The optional restriction part is covered by clause 15.8.

In-line modified template:

```
[ Type ":" ] modifies TemplateRefWithParList ":=" TemplateBody
```

Semantic Description

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

The modifications occur in a linked fashion eventually tracing back to the original template. If a template field and its corresponding value or matching symbol is specified in the modified template, then the specified value or matching symbol replaces the one specified in the parent template. If a template field and its corresponding value or matching symbol is not specified in the modified template, then the value or matching symbol in the parent template shall be used. When the field to be modified is nested within a template field which is a structured field itself, no other field of the structured field is changed apart from the explicitly denoted one(s).

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

Formal value or template parameters of modified templates inherit the default value or respectively template of the corresponding parameter of their parent templates only, if this is denoted by the dash (don't change) symbol at the place of the parameters' default value or respectively template.

Modified templates may also be restricted. Template restrictions are specified in clause 15.8.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

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

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

   1) the derived template shall not omit parameters defined at any of the modification steps between the base template and the actual modified template;

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

   3) if the dash (don't change) symbol is used at the place of a default value or default template, the corresponding parameter of the parent template shall have a valid default value or default template, either assigned directly or inherited. If not, this shall cause an error.

c) Restrictions on referencing elements of templates or template fields are described in clause 15.6: for modified templates the rules for the left hand side of assignments apply.

d) Limitations on template restrictions described in clause 15.8 shall apply.

Examples

EXAMPLE 1:

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

```
    field3 := true
}
```

EXAMPLE 2: Modified record of template

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

EXAMPLE 3: Modified in-line template

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

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

EXAMPLE 4: Modified parameterized template

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

// then a modification could be
    template MyRecordType MyTemplate2(integer MyPar) modifies MyTemplate1 :=
{   // field1 is parameterized in Template1 and remains also parameterized in Template2
    field2 := "A modified string"
}
```

EXAMPLE 5: Default values of modified parameterized templates

```
// Given
template MyRecordType MyTemplate11 (integer p_int := 5 ):= {
  // p_int has the default value 5
    field1 := p_int,
    field2 := "A string",
    field3 := true
}

// then possible template modifications are
template MyRecordType MyTemplate12(integer p_int) modifies MyTemplate11 := {
  // p_int had a default value in MyTemplate11 but has none in this template
  field2 := "B string"
}

template MyRecordType MyTemplate13(integer p_int := 0) modifies MyTemplate12 := {
  // p_int has the default value 0
  // no change is made to the template's content, but only to the default value of p_int
}

template MyRecordType MyTemplate14(integer p_int := - ) modifies MyTemplate13 := {
  // p_int inherits the default value 0 from its parent MyTemplate13
  field2 := "C string"
}

template MyRecordType MyTemplate15(integer p_int := - ) modifies MyTemplate14 := {
  // p_int inherits the default value 0 from MyTemplate13 via MyTemplate14
  field2 := "D string"
}

template MyRecordType MyTemplate16(integer p_int) modifies MyTemplate15 := {
  // p_int has no default value
}

template MyRecordType MyTemplate17(integer p_int := - ) modifies MyTemplate16 := {
  // causes an error as p_int has no default value in the parent template MyTemplate16
  field2 := "E string"
}
```

15.6 Referencing elements of templates or template fields

This clause defines rules and restrictions when referencing elements of templates or template fields.

15.6.1 Referencing individual string elements

It is not allowed to reference individual string elements inside templates or template fields. Instead, the **substr** function (see clause C.34) shall be used.

EXAMPLE:

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

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

15.6.2 Referencing **record** and **set** fields

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

a) Omit, AnyValueOrNone, value lists and complemented lists: referencing a subfield within a structured field to which Omit, AnyValueOrNone, a value list or a complemented list is assigned, at the right hand side of an assignment, shall cause an error.
When referencing a subfield within a structured field to which AnyValueOrNone or omit is assigned, at the left hand side of an assignment, the structured field is implicitly set to be present, it is expanded recursively up to and including the depth of the referenced subfield. During this expansion an AnyValue shall be assigned to mandatory subfields and AnyValueOrNone shall be assigned to optional subfields. After this expansion the value or matching mechanism at the right hand side of the assignment shall be assigned to the referenced subfield.
When referencing a subfield within a structured field to which value lists or complemented value lists are assigned, at the left hand side of an assignment, shall cause an error.

EXAMPLE 1:

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

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

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

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

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

t_R2 := t_R1.f2;
t_R2 := t_R1.f2.g2;
```

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

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

EXAMPLE 2:

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

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

15.6.3 Referencing **`record of`** and **`set of`** elements

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

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

EXAMPLE 1:

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

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

EXAMPLE 2:

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

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

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

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

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

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

EXAMPLE 3:

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

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

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

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

15.6.4 Referencing signature parameters

While signature templates do not allow referencing their parameters directly (e.g. using dot notation), such a reference is possible when modifying a signature template. However, there can be a matching mechanism assigned to the signature template. This clause provides rules for such cases.

a) Value lists and complemented lists: referencing a parameter of a signature template to which a value list or a complemented list is assigned, at the left hand side of an assignment, shall cause an error.

EXAMPLE 1:

```
signature MySignature(in integer par1, in integer par2);
template MySignature t_mySign1 := ({ par1 := 1, par2 := 2 }, { par1 := 2, par2 := 1 });
template MySignature t_mySign2 modifies t_mySign1 := { par1 := ? };
// shall cause an error as t_mySign1 contains a value list template
```

b) AnyValue: when referencing a parameter within a signature to which AnyValue is assigned, at the left hand side of an assignment, the signature template is implicitly expanded to the parameter level. During this expansion an AnyValue shall be assigned to all parameters of the template. After this expansion the value or matching mechanism at the right hand side of the assignment shall be assigned to the referenced parameter.

EXAMPLE 2:

```
template MySignature t_mySign3 := ?;
template MySignature t_mySign4 modifies t_mySign3 := { par1 := 3 };
// t_mySign3 is expanded to { par1 := ?, par2 := ? }, then 3 is assigned to par1,
// thus t_mySign4 will be { par1 := 3, par2 := ? }
```

15.7 Template matching mechanisms

Generally, matching mechanisms are used to replace values of single template fields or to replace even the entire contents of a template. Matching mechanisms may also be used in-line (see clause 15.4*)*.

Matching mechanisms are arranged in four groups:

- specific values;
- special symbols that can be used *instead* of values;
- special symbols that can be used *inside* values;
- special symbols which describe *attributes* of values;

Some of the mechanisms may be used in combination.

The supported matching mechanisms and their associated symbols (if any) and the scope of their application are shown in table 11. The left-hand column of this table lists all the TTCN-3 types to which these matching mechanisms apply. A full description of each matching mechanism can be found in annex B.

Table 11: TTCN-3 Matching Mechanisms

Used with values of	Value	Instead of values									Inside values			Attributes	
	SpecificValue	Omit	ComplementedList	ValueList	AnyValue (?)	AnyValueOrNone (*)	Range	Superset	Subset	Pattern	AnyElement (?)	AnyElementsOrNone (*)	Permutation	LengthRestriction	IfPresent
boolean	Yes	Yes[1]	Yes	Yes	Yes	Yes[1]									Yes[1]
integer	Yes	Yes[1]	Yes	Yes	Yes	Yes[1]	Yes								Yes[1]
float	Yes	Yes[1]	Yes	Yes	Yes	Yes[1]	Yes								Yes[1]
bitstring	Yes	Yes[1]	Yes	Yes	Yes	Yes[1]					Yes	Yes		Yes	Yes[1]
octetstring	Yes	Yes[1]	Yes	Yes	Yes	Yes[1]					Yes	Yes		Yes	Yes[1]
hexstring	Yes	Yes[1]	Yes	Yes	Yes	Yes[1]					Yes	Yes		Yes	Yes[1]
character strings	Yes	Yes[1]	Yes	Yes	Yes	Yes[1]	Yes			Yes	Yes[2]	Yes[2]		Yes	Yes[1]
record	Yes	Yes[1]	Yes	Yes	Yes	Yes[1]									Yes[1]
record of	Yes	Yes[1]	Yes	Yes	Yes	Yes[1]					Yes	Yes	Yes	Yes	Yes[1]
array	Yes	Yes[1]	Yes	Yes	Yes	Yes[1]					Yes	Yes		Yes	Yes[1]
set	Yes	Yes[1]	Yes	Yes	Yes	Yes[1]									Yes[1]
set of	Yes	Yes[1]	Yes	Yes	Yes	Yes		Yes	Yes		Yes	Yes		Yes	Yes[1]
enumerated	Yes	Yes[1]	Yes	Yes	Yes	Yes[1]									Yes[1]
union	Yes	Yes[1]	Yes	Yes	Yes	Yes[1]									Yes[1]
anytype	Yes	Yes[1]	Yes	Yes	Yes	Yes[1]									Yes[1]

NOTE 1: Can be assigned to templates, however when used shall be applied to optional fields of record and set types only (without restriction on the type of that field).
NOTE 2: Have matching mechanism meaning within character patterns only.

15.7.1 Specific values

Specific values are the basic matching mechanism of TTCN-3 templates. Specific values in templates are expressions which do not contain any matching mechanisms.

Syntactical Structure

```
SingleExpression
```

Semantic Description

The matching mechanism for a specific value is an expression that evaluates to a specific value.

For further details please refer to clause 6 and to annex B.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) See the restrictions given in table 11 and in annex B.

Examples

```
MyPCO.receive(charstring:"abcxyz");
MyPCO.receive('AAAA'O);
```

15.7.2 Special symbols that can be used instead of values

These matching mechanisms can be used to characterize a set of values.

Syntactical Structure

```
omit |
"(" { TemplateInstance [","] } ")" |
complement "(" { TemplateInstance [","] } ")" |
"?" |
"*" |
"(" ( ConstantExpression | -infinity ) ".." ( ConstantExpression | infinity ) ")" |
superset "(" { ConstantExpression [","] } ")" |
subset "(" { ConstantExpression [","] } ")" |
pattern Cstring
```

Semantic Description

The matching mechanisms for special symbols that can be used *instead* of values are:

- **omit:** the optional field, in which it is used, is not present;
- **(…):** a list of values or templates;
- **complement (…):** complement of a list of values or templates;
- **?:** wildcard for any value;
- ***:** wildcard for any value or no value at all, i.e. the field is not present;
- **(***lowerBound* **..** *upperBound***):** a range of integer or float values between and including the lower- and upper bounds;
- **superset:** at least all of the elements listed, i.e. possibly more;
- **subset:** at most the elements listed, i.e. possibly less;
- **pattern:** a charstring or universal charstring that matches this format.

For further details please refer to annex B.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) See the restrictions given in table 11 and in annex B.

Examples

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

15.7.3 Special symbols that can be used inside values

These matching mechanisms allow to characterize value sets by varying values inside.

Syntactical Structure

```
…"?"… |
…"*"… |
…permutation "(" { ( TemplateBody | "?" | "*" )[","] } ")"…
```

Semantic Description

The matching mechanisms for special symbols that can be used *inside* values are:

- **?**: wildcard for any single element in a string, array, **`record of`** or **`set of;`**
- *****: wildcard for any number of consecutive elements in a string, array, **`record of`** or **`set of`**, or no element at all (i.e. an omitted element);
- **permutation**: all of the elements listed but in an arbitrary order (note, that ? and * are also allowed as elements of the permutation list).

For further details please refer to annex B.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) See the restrictions given in table 11 and in annex B.

Examples

```
template bitstring b := '10???'B;       // where each "?" may either be 0 or 1
type record of integer RI;
template RI ri := {1, ?, 3}             // where ? may be any integer value
```

15.7.4 Special symbols which describe attributes of values

These matching mechanisms define properties of values.

Syntactical Structure

```
length "(" ConstantExpression [ ".." ( ConstantExpression | infinity ) ] ")" [ ifpresent ] |
ifpresent
```

Semantic Description

The matching mechanisms which describe *attributes* of values are:

- **length**: restrictions for string length of string types and the number of elements for **`record of`**, **`set of`** and arrays;
- **ifpresent**: for matching of optional field values (if not omitted).

For further details please refer to annex B.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) See the restrictions given in table 11 and in annex B.

Examples

```
type record R {
    record of integer ri optional
}
template R r:=
{
    ri := * length (1 .. 6) ifpresent       // any value containing 1, 2, 3, 4,
                                            // 5 or 6 elements, provided it is present
}
```

15.8 Template Restrictions

Template restrictions allow to restrict the matching mechanisms that can be used with a template. Template restrictions are applicable to template definitions and template variables, formal template parameters, and return template types of functions. Template restrictions can be applied equally to message and signature templates.

Syntactical Structure

```
template "(" ( omit | present | value ) ")" Type
```

Semantic Description

The restrictions mean in case of:

- **(omit)** the template shall resolve to a value matching mechanism (i.e. the fields of it shall resolve to a specific value or omit, and the whole template may also resolve to omit). Such a template can be used to define a field of a record and set template and the latter one could still be used in a **send** statement.

- **(value)** the template shall resolve to a specific value (i.e. the fields of it shall resolve to a specific value or omit, but the whole template shall not resolve to omit). It can be used to define a mandatory field of a record or set template and the latter one could still be used in a **send** statement.

- **(present)** the template as a whole shall not resolve to matching mechanisms that match omit (i.e. its fields may contain any of the matching mechanisms or matching attributes). Such a template can be used to define a mandatory field of a record or set template.

NOTE: Template restrictions allow TTCN-3 tools to check more easily at compile time whether templates and matching expressions are used correctly. Whether the checks are performed at compile time and invalid code is rejected or whether the checks are performed at execution time and dynamic errors are raised, is outside the scope of the present document.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) Matching mechanisms can be used within restricted templates according to table 12.

Table 12: Using matching mechanisms with restricted templates

Used with template restriction	Value		Instead of values								Inside values			Attributes	
	SpecificValue	OmitValue	ComplementedList	ValueList	AnyValue (?)	AnyValueOrNone (*)	Range	Superset	Subset	Pattern	AnyElement (?)	AnyElementsOrNone (*)	Permutation	LengthRestriction	IfPresent
omit	Yes	Yes													
value	Yes	Note													
present	Yes	Note		Yes	Yes	Note	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Note
NOTE: It is allowed to use the matching mechanism in fields of the template, but the template as a whole shall not resolve to this matching mechanism.															

b) Restricted and unrestricted templates can be used as actual parameters of formal template parameters or assigned to template variables according to table 13.

Table 13: Restrictions of formal and actual template parameters

	Actual parameter/right hand side of an expression	value	template (omit)	template (value)	template (present)	template
Formal parameter/-left hand side of an expression						
template(omit)		Yes	Yes	Yes	(see note)	(see note)
template(value)		Yes	(see note)	Yes	(see note)	(see note)
template(present)		Yes	(see note)	Yes	Yes	(see note)
template		Yes	Yes	Yes	Yes	Yes
NOTE: These restrictions are related to the content of the actual parameter or right hand side expression and not to the definition of the entities used. Which cases are checked at compile time and which ones at runtime is a tool implementation issue.						

c) A restricted, modified template has to have the same or more restrictive restriction as the base template. A restricted parameter of a modified template has to have the same or a more restrictive restriction as the corresponding parameter of the base template. The allowed restrictions are listed in table 14.

Table 14: Restricting modified templates

Restriction in base template	Allowed restrictions in modified template
template	template, template(present), template(omit), template(value)
template(present)	template(present), template(value)
template(omit)	template(omit), template(value)
template(value)	template(value)

Examples

```
// definitions of restricted templates
type record ExampleType {
    integer a,
    boolean b optional
}

template(omit) ExampleType exampleOmit := omit;
template(omit) ExampleType exampleOmitValue:= { 1, true };
template(omit) ExampleType exampleOmitAny := ?;           // incorrect

template(value) ExampleType exampleValueomit := omit;     // incorrect
template(value) ExampleType exampleValue := { 1, true };
template(value) ExampleType exampleValueOptional := { 1, omit };
                                                          // omit assigned to a field is correct

template(present) ExampleType examplePresent := {1, ?};
template(present) ExampleType examplePresentIfpresent := { 1, true } ifpresent;
                                                          // incorrect
template(present) ExampleType examplePresentAny := ?;


// restricted template usage
var template ExampleType (omit) v_omit;
var template ExampleType (present) v_present;
var template ExampleType (value) v_value;

v_omit := exampleOmit;
v_omit := exampleValueOptional;
v_omit := examplePresentAny;     // incorrect, not a specific value

v_present := exampleOmit;        // incorrect, must not be omit
v_present := examplePresent;

v_value := exampleOmit;          // incorrect, must not be omit
v_value := examplePresentAny;    // incorrect, must be a single value

// template modification
template (present) ExampleType exampleBase( template (omit) boolean p) := { ?, p };

//correct, template and its parameter are more restrictive
template (value) ExampleType exampleModified( template (value) boolean p)
    modifies exampleBase := { a := 1 };
//incorrect, modified template is less restrictive
template ExampleType exampleModified( template (value) boolean p)
    modifies exampleBase := { a := 1 };
//incorrect, parameter of modified template is less restrictive
template (present) ExampleType exampleModified( template (present) boolean p)
    modifies exampleBase := { a := 1 };
```

15.9 Match Operation

The **match** operation allows to compare a value (specified in form of an expression) with a template.

Syntactical Structure

```
match "(" Expression "," TemplateInstance ")"
```

Semantic Description

The **match** operation returns a boolean value. If the types of the template and the value (specified in form of an expression) are not compatible (see clause 6.3) the operation returns **false**. If the types are compatible, the return value of the **match** operation indicates whether the value matches the specified template.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The expression-parameter of the **match** operation shall not evaluate to a template, i.e. the **match** operation cannot be used to compare two templates.

Examples

```
template integer LessThan10 := (-infinity..9);
:
MyPort.receive(integer:?) -> value RxValue;
if( match( RxValue, LessThan10)) { … }
// true if the actual value of Rxvalue is less than 10 and false otherwise
:
```

15.10 Valueof Operation

The **valueof** operation allows to return the value specified within a template. The returned value can be assigned to a variable, may be used in expressions, as an actual value parameter, etc.

Syntactical Structure

valueof "(" *TemplateInstance* ")"

Semantic Description

The **valueof** operation returns the value of a template instance.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The template shall be completely initialized and resolve to a specific value.

Examples

EXAMPLE 1:

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

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

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

EXAMPLE 2:

```
function MyFunc() {
  var template integer vt_int := omit;
  //is ok, but to be used for optional record or set fields only
  var integer v_int := valueof(vt_int)
  //causes an error as omit is not a value and shall not be an argument of valueof
  :
}
```

15.11 Concatenating templates of string and list types

Templates of string and list types (bitstring, octetstring, hexstring, charstring, universal charstring, record of, set of, and array) can be concatenated from several single (inline) templates using the concatenation operation. Each single template shall have the same root type. The single templates of binary string and list types shall contain only the matching mechanisms specific values, AnyValue or AnyValueOrNone constrained to a fixed length, AnyElement, or AnyElementsOrNone possibly constrained with a length attribute for list types. The concatenation results in the sequential concatenation of the single templates from left to right, with one exception: matching symbols AnyValue, AnyValueOrNone, AnyElement and AnyElementsOrNone constrained to a fixed length N shall be replaced by N AnyElement matching symbols before concatenation. The concatenation shall be performed completely before using the resulting template (e.g. for assignment or matching) and the result shall be type-compatible with the place of its use.

NOTE 1: Inline templates used for the concatenation need not be valid templates of the result type (e.g. odd number of hexadecimal digits are allowed in an octetstring template concatenation), but the resulting template has to be a valid template.

NOTE 2: See also concatenation of character string patterns in clause B.1.5.

EXAMPLE 1: Composing templates of string types

```
template charstring t_Mychar1 := "ABC" & "DE*" & "F?";
    // results in the template "ABCDE*F?"
    // please note that "*" and "?" denote the characters "*" and "?"
template charstring t_Mychar2 := "ABC" & * length(2) & "EF";
    // causes an error as for character string types only
    // specific values are allowed

template bitstring t_Mybit := '010'B & '*'B & '1?1'B;
    // results in the template '010*1?1'B
template octetstring t_Myoct1 := 'ABC'O & 'D*'O & '?EF'O;
    // results in the template 'ABCD*?EF'O
template octetstring t_Myoct2 := 'ABCD'O & ? length (2) & 'EF'O;
    // results in the template 'ABCD??EF'O
    // (i.e. a 5 octets i.e. 10 hexadecimal digits long value)

template octetstring t_Myoct := 'ABCD'O & '?'O & '?E'O;
    // causes an error, the resulting template shall be a legal value
    // (if composed, 'ABCD??E'O would denote 9 hexadecimal digits, but the length
    // should be an even number of digits)
template charstring t_MycharWrong := "ABC" & * length(1..2) & "E?F";
    // causes an error, the length attribute shall be of fixed length

template hexstring t_MyhexPar (integer N):= 'ABC'H & ? length(N) & 'E?F'H;
function MyFunc() runs on MyCompType {
  var integer v_int := 3;
  var template hexstring vt_hstring;
  :
  vt_hstring := 'ABC'H & ? length(v_int) & 'E?F'H;
    //results in the template 'ABC???E?F'H
  P.receive (t_MyhexPar(4));
    //actual content of t_MyhexPar is "ABC????E?F"
}
```

EXAMPLE 2: Composing templates of list types

```
type record of charstring RecofChar;
type set of integer SetofInt;

template RecofChar t_MyRecofChar := {"ABC"} & {"D?", "EF"};
    // results the template {"ABC", "D", "EF" }
template SetofInt t_MySetofInt := { 1, 2 } & ? length(2) & { 3, 4 };
    // results the template {1, 2, ?, ?, 3, 4 }
template RecofInt t_MyRecofInt := { 1, 2 } & { * length(2), 3, 4 };
    // results the template {1, 2, ?, ?, 3, 4 }

template RecofChar t_MyRecofCharWrong:= {"ABC"} & ? length(1..2) & {"EF"};
    // causes an error, the length attribute shall denote a fixed length

template RecofChar t_MyRecofCharPar (integer N):= { "ABC" }, ? * length(N) & { "EF" };
function MyFunc() runs on MyCompType{
  var integer v_int := 3;
  var template RecofChar vt_recofChar;
  :
  vt_recofChar :=  { "ABC" } & ? length(v_int) & { "EF" };
    //results the template { "ABC", ?, ?, ?, "EF" }
  P.receive ( t_MyRecofCharPar(4) );
    //actual content of t_MyRecofCharPar is { "ABC", ?, ?, ?, ?, "EF" }
}
```

16 Functions, altsteps and testcases

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

16.1 Functions

Functions are used in TTCN-3 to express test behaviour, to organize test execution or to structure computation in a module, for example, to calculate a single value, to initialize a set of variables or to check some condition.

Syntactical Structure

```
function FunctionIdentifier
"(" [ { ( FormalValuePar | FormalTimerPar | FormalTemplatePar | FormalPortPar ) [","] } ] ")"
[ runs on ComponentType ]
[ return [ template ] Type ]
StatementBlock
```

Semantic Description

Functions are portions of TTCN-3 behaviour, which perform a specific task and are relatively independent of the remaining behaviour.

Functions may return a value or a template. Value return is denoted by the **return** keyword followed by a type expression. Template return is denoted by the **return template** keywords followed by an optional restriction and a type expression. Execution of a **return** statement in the body of the function causes the function to terminate and to return the result to the location of the call of the function.

The behaviour of a function can be defined by using statements and operations described in clauses 18 to 25 and clause 26. If a function uses variables, constants, timers and ports that are declared in a component type definition, the component type shall be referenced using the **runs on** keywords in the function header. The one exception to this rule is if all the necessary component-wide information is passed in the function as parameters.

Functions may be parameterized.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

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

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

NOTE 1: The restrictions concerning the **runs on** clause are only related to functions and altsteps and not to test cases.

c) Functions used in the control part of a TTCN-3 module shall have no **runs on** clause.

NOTE 2: Nevertheless, functions used in the control part are allowed to execute test cases.

d) The rules for formal parameter lists shall be followed as defined in clause 5.4.

e) For **return template** statements the restrictions specified in clause 15 shall apply.

f) Template **return** can be restricted to the matching mechanisms specific value and **omit**, see clause 5.4.1.2.

g) A **return** statement in a value returning function shall always have a value expression compatible to the type specified in the function header return clause.

h) A **return** statement in a template returning function shall always have a template expression or template instance compatible to the type specified in the function header return clause. If the **return** clause has a template restriction, this restriction must be adhered to by the returned template.

i) If the function header includes a **return** clause the function, when terminating, shall do so by executing a **return** statement. The function will cause a test case error if it terminates (i.e. reaches the end of the function body) without executing a **return** statement.

Examples

EXAMPLE 1: Function with return

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

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

EXAMPLE 2: Function with template return

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

EXAMPLE 3: Function with runs on clause

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

EXAMPLE 4: Parameterized function

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

EXAMPLE 5: Function without return statement

```
function MyFunction5(inout integer MyPar1) return integer {
    if (MyPar1 > 5) {
      MyPar1 := 5;
      return MyPar1;
    }
    // in case of MyPar1 <= 5, MyFunction5 doesn't terminate in a return statement
    // and will cause a test case error
}
```

16.1.1 Invoking functions

A function is invoked by referring to its name and providing the actual list of parameters.

Syntactical Structure

```
FunctionRef "(" [ { ActualPar [","] } ] ")"
```

Semantic Description

A function invocation results in the execution of the statement block of the invoked function. The invoked function is performed by the test component invoking it. Actual parameters are passed into the statement block. If the function returns (upon termination and potentially with a return value), the test components continues its behaviour right after the function invocation.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) Functions that do not return values shall be invoked directly. Functions that return values may be invoked directly or inside expressions.

b) The rules for actual parameter lists shall be followed as defined in clause 5.4.

c) Special restrictions apply to functions bound to test components using the **`start`** test component operation. These restrictions are described in clause 21.3.2.

d) When invoking a function, the compatibility to the test component type of the invoking test component as described in clause 6.3.3 need to be fulfilled.

e) Restrictions on invoking functions from specific places are described in clause 16.1.4.

Examples

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

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

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

16.1.2 Predefined functions

TTCN-3 contains a number of predefined (built-in) functions that need not be declared before use. These are summarized in table 15.

Table 15: List of TTCN-3 predefined functions

Category	Function	Keyword
Conversion functions	Convert **integer** value to **charstring** value	`int2char`
	Convert **integer** value to **universal charstring** value	`int2unichar`
	Convert **integer** value to **bitstring** value	`int2bit`
	Convert **integer** value to enumeration	`int2enum`
	Convert **integer** value to **hexstring** value	`int2hex`
	Convert **integer** value to **octetstring** value	`int2oct`
	Convert **integer** value to **charstring** value	`int2str`
	Convert **integer** value to **float** value	`int2float`
	Convert **float** value to **integer** value	`float2int`
	Convert **charstring** value to **integer** value	`char2int`
	Convert **charstring** value to **octetstring** value	`char2oct`
	Convert **universal charstring** value to **integer** value	`unichar2int`
	Convert **bitstring** value to **integer** value	`bit2int`
	Convert **bitstring** value to **hexstring** value	`bit2hex`
	Convert **bitstring** value to **octetstring** value	`bit2oct`
	Convert **bitstring** value to **charstring** value	`bit2str`
	Convert **hexstring** value to **integer** value	`hex2int`
	Convert **hexstring** value to **bitstring** value	`hex2bit`
	Convert **hexstring** value to **octetstring** value	`hex2oct`
	Convert **hexstring** value to **charstring** value	`hex2str`
	Convert **octetstring** value to **integer** value	`oct2int`
	Convert **octetstring** value to **bitstring** value	`oct2bit`
	Convert **octetstring** value to **hexstring** value	`oct2hex`
	Convert **octetstring** value to **charstring** value	`oct2str`
	Convert **octetstring** value to **charstring** value, version II	`oct2char`
	Convert **charstring** value to **integer** value	`str2int`
	Convert **charstring** value to **hexstring** value	`str2hex`
	Convert **charstring** value to **octetstring** value	`str2oct`
	Convert **charstring** value to **float** value	`str2float`
	Convert enumeration to **integer** value	`enum2int`
Length/size functions	Return the length of a value or template of any string type, **record of**, **set of** or **array**	`lengthof`
	Return the number of elements in a value or a template of a **record** or **set**	`sizeof`
Presence checking functions	Determine if an optional field in a **record** or **set** value or template is present	`ispresent`
	Determine which choice has been selected in a **union** value or template	`ischosen`
	Determine if a template evaluates to a concrete value	`isvalue`
String/List handling functions	Returns part of the input string matching the specified pattern group within a character pattern	`regexp`
	Returns the specified portion of the input string/list value or template	`substr`
	Replaces a substring of a string with or inserts the input string into a string, and similarly for lists	`replace`
Codec functions	Encode a value into a bitstring	`encvalue`
	Decode a bitstring into a value	`decvalue`
Other functions	Generate a random float number	`rnd`
	Returns the name of the currently executing test case	`testcasename`

Syntactical Structure

```
int2char "(" SingleExpression ")" |
int2unichar "(" SingleExpression ")" |
int2bit "(" SingleExpression "," SingleExpression ")" |
int2enum "(" SingleExpression "," SingleExpression ")" |
int2hex "(" SingleExpression "," SingleExpression ")" |
int2oct "(" SingleExpression "," SingleExpression ")" |
int2str "(" SingleExpression ")" |
int2float "(" SingleExpression ")" |
float2int "(" SingleExpression ")" |
char2int "(" SingleExpression ")" |
char2oct "(" SingleExpression ")" |
unichar2int "(" SingleExpression ")" |
bit2int "(" SingleExpression ")" |
bit2hex "(" SingleExpression ")" |
bit2oct "(" SingleExpression ")" |
bit2str "(" SingleExpression ")" |
hex2int "(" SingleExpression ")" |
hex2bit "(" SingleExpression ")" |
hex2oct "(" SingleExpression ")" |
hex2str "(" SingleExpression ")" |
oct2int "(" SingleExpression ")" |
oct2bit "(" SingleExpression ")" |
oct2hex "(" SingleExpression ")" |
oct2str "(" SingleExpression ")" |
oct2char "(" SingleExpression ")" |
str2int "(" SingleExpression ")" |
str2hex "(" SingleExpression ")" |
str2oct "(" SingleExpression ")" |
str2float "(" SingleExpression ")" |
enum2int "(" SingleExpression ")" |
lengthof "(" TemplateInstance ")" |
sizeof "(" TemplateInstance ")" |
ispresent "(" TemplateInstance ")" |
ischosen "(" TemplateInstance ")" |
isvalue "(" TemplateInstance ")" |
regexp "(" TemplateInstance"," TemplateInstance"," SingleExpression ")" |
substr "(" TemplateInstance "," SingleExpression "," SingleExpression ")" |
replace "(" SingleExpression "," SingleExpression "," SingleExpression "," SingleExpression ")" |
encvalue "(" TemplateInstance ")" |
decvalue "(" SingleExpression "," SingleExpression ")" |
rnd "(" [ SingleExpression ] ")" |
testcasename "()"
```

Semantic Description

The description of predefined functions is given in annex C.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) When a predefined function is invoked:

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

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

   3) all actual parameters shall be initialized with the exception of the actual parameter passed to the isvalue predefined function, which may be uninitialized.

b) Restrictions on invoking functions from specific places are described in clause 16.1.4.

Examples

```
var hexstring h:= bit2hex ('111010111'B);
var octetstring o:= substr ('01AB23CD'O, 1, 2);
```

16.1.3 External functions

A function may be defined within a module or be declared as being defined externally (i.e. **external**).

Syntactical Structure

```
external function ExtFunctionIdentifier
"(" [ { ( FormalValuePar | FormalTimerPar | FormalTemplatePar | FormalPortPar ) [","] } ] ")"
[ return Type ]
```

Semantic Description

For an external function only the function interface has to be provided in the TTCN-3 module. The realization of the external function is outside the scope of the present document.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) External functions are not allowed to contain port, timer or default handling operations.

b) External functions are not allowed to return templates.

c) Restrictions on invoking functions from specific places are described in clause 16.1.4.

Examples

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

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

16.1.4 Invoking functions from specific places

Value returning functions can be called in communication operations (in templates, template fields, in-line templates, or as actual parameters), in guards of alt statements or altsteps (see clause 20.2), and in initializations of altstep local definitions (see clause 16.2). To avoid side effects that cause changing the state of the component or the actual snapshot and to prevent different results of subsequent evaluations on an unchanged snapshot, the following operations shall not be used in functions called in the cases specified above:

a) All component operations, i.e. **create**, **start** (component), **stop** (component), **kill**, **running** (component), **alive, done** and **killed** (see notes 1, 3, 4 and 6).

b) All port operations, i.e. **start** (port), **stop** (port), **halt**, **clear**, **send**, **receive**, **trigger**, **call**, **getcall**, **reply**, **getreply**, **raise**, **catch**, **check**, **connect**, **map** (see notes 1, 2, 3 and 6).

c) The **action** operation (see notes 2 and 6).

d) All timer operations, i.e. **start** (timer), **stop** (timer), **running** (timer), **read**, **timeout** (see notes 4 and 6).

e) Calling external functions (see notes 4 and 6).

f) Calling the **rnd** predefined function (see notes 4 and 6).

g) Changing of component variables, i.e. using component variables on the left-hand side of assignments, and in the instantiation of **out** and **inout** parameters (see notes 4 and 6).

h) Calling the **setverdict** operation (see notes 4 and 6).

i) Activation and deactivation of defaults, i.e. the **activate** and **deactivate** statements (see notes 5 and 6).

j) Calling functions with **out** or **inout** parameters (see notes 7 and 8).

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

NOTE 2: The use of operations **send**, **call**, **reply**, **raise**, and **action** causes an error, i.e. all communication are to be made explicit and not as a side-effect of another communication operation or the evaluation of a snapshot.

NOTE 3: The use of operations **map**, **unmap**, **connect**, **disconnect**, **create** causes an error, i.e. all configuration operations are to be made explicit, and not as a side-effect of a communication operation or the evaluation of a snapshot.

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

NOTE 5: The use of operations **activate** and **deactivate** causes an error because they modify the set of defaults that is considered during the evaluation of the current snapshot.

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

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

NOTE 8: Using **out** or **inout** parameters causes an error because it may lead to different results of subsequent evaluations of the same snapshot.

16.2 Altsteps

TTCN-3 uses altsteps to specify default behaviour or to structure the alternatives of an **alt** statement.

Syntactical Structure

```
altstep AltstepIdentifier
"(" [ { ( FormalValuePar | FormalTimerPar | FormalTemplatePar | FormalPortPar ) [","] } ] ")"
[ runs on ComponentType ]
"{"
     { ( VarInstance | TimerInstance | ConstDef | TemplateDef ) [";"] }
     AltGuardList
"}"
```

Semantic Description

Altsteps are scope units similar to functions. The altstep body defines an optional set of local definitions and a set of alternatives, the so-called *top alternatives*, that form the altstep body. The syntax rules of the top alternatives are identical to the syntax rules of the alternatives of **alt** statements.

The behaviour of an altstep can be defined by using the program statements and operations summarized in clause 18. Altsteps may invoke functions and altsteps or activate altsteps as defaults.

Altsteps may be parameterized as defined in clause 5.4.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The local definitions of an altstep shall be defined before the set of alternatives.

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

c) If an altstep includes port operations or uses component variables, constants or timers the associated component type shall be referenced using the **runs on** keywords in the altstep header. The one exception to this rule is if all ports, variables, constants and timers used within the altstep are passed in as parameters.

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

e) An altstep that is activated as a default shall only have **in** value or template parameters, port parameters, and timer parameters. An altstep that is only invoked as an alternative in an **alt** statement or as stand-alone statement in a TTCN-3 behaviour description may have **in**, **out** and **inout** parameters. The rules for formal parameter lists shall be followed as defined in clause 5.4.

Examples

EXAMPLE 1: Parameterized altstep with runs on clause

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

// Altstep definition using PCO1, PCO2, MyIntVar and MyTimer of MyComponentType
altstep AltSet_A(in integer MyPar1) runs on MyComponentType {
    [] PCO1.receive(MyTemplate(MyPar1, MyIntVar) {
           setverdict(inconc);
       }
    [] PCO2.receive {
           if (MyPar1 != 0) {
               repeat
           }
           else {
               break
           }
       }
    [] MyTimer.timeout {
        setverdict(fail);
           stop
       }
}
```

EXAMPLE 2: Altstep with local definitions

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

16.2.1 Invoking altsteps

The invocation of an altstep is always related to an **alt** statement. The invocation may be done either implicitly by the default mechanism (see clause 21) or explicitly by a direct call within an **alt** statement (see clause 20.2).

Syntactical Structure

```
AltstepRef "(" [ { ActualPar [","] } ] ")"
```

Semantic Description

The invocation of an altstep causes no new snapshot and the evaluation of the top alternatives of an altstep is done by using the actual snapshot of the **alt** statement from which the altstep was called.

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

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

An explicit call of an altstep within an **alt** statement looks syntactically like a function invocation as an alternative. When an altstep is called explicitly within an **alt** statement, the next alternative to be checked is the first alternative of the **altstep**. The alternatives of the **altstep** are checked and executed the same way as alternatives of an **alt** statement (see clause 20.1) with the exception that no new snapshot is taken when entering the **altstep**. An unsuccessful termination of the altstep (i.e. all top alternatives of the **altstep** have been checked and no matching branch is found) causes the evaluation of the next alternative or invocation of the default mechanism (if the explicit call is the last alternative of the **alt** statement). A successful termination may cause either the termination of the test component, i.e. the altstep ends with a **stop** statement, or a new snapshot and re-evaluation of the **alt** statement, i.e. the altstep ends with **repeat** (see clause 20.2) or a continuation immediately after the **alt** statement, i.e. the execution of the selected top alternative of the altstep ends with a **break** statement (see clause 19.12) or without explicit **repeat** or **stop**.

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

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) When invoking an altstep, the compatibility of the test component type of the invoking test component and of the altstep runs on clause (as described in clause 6.3.3) need to be fulfilled.

b) Further restrictions on invoking altsteps in the activate statement are given in clause 20.5.2.

Examples

EXAMPLE 1: Implicit invocation of an altstep via a default activation

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

EXAMPLE 2: Explicit invocation of an altstep within an alt statement

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

EXAMPLE 3: Explicit, stand-alone invocation of an altstep

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

//is a shorthand for

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

16.3 Test cases

A test case is complete and independent specification of the actions required to achieve a specific test purpose. It typically starts in a stable testing state and ends in a stable testing state. It may involve one or more consecutive or concurrent connections to the SUT. The test case shall be complete in the sense that it is sufficient to enable a test verdict to be assigned unambiguously to each potentially observable test outcome (i.e. sequence of test events). The test case shall be independent in the sense that it shall be possible to execute the derived executable test case in isolation from other such test cases.

In TTCN-3, test cases are a special kind of function. Test cases define the behaviours, which have to be executed to check whether the SUT passes a test or not. This behaviour is performed by the MTC which is automatically created when a test case is being executed.

Syntactical Structure

```
testcase TestcaseIdentifier
"(" [ { ( FormalValuePar | FormalTemplatePar) [","] } ] ")"
runs on ComponentType
[ system ComponentType ]
StatementBlock
```

Semantic Description

A test case is considered to be a self-contained and complete specification that checks a test purpose. The result of a test case execution is a test verdict.

A test case header has two parts:

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

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

The behaviour of a test case can be defined by using the program statements and operations described in clause 18.

Test cases may be parameterized as described in clause 5.4. Test cases can be executed in the control part of a module (see clause 26).

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The rules for formal parameter lists shall be followed as defined in clause 5.4.

b) Test cases may only be invoked with an execute statement in a module control part as defined in clause 26.

Examples

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

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

17 Void

18 Overview of program statements and operations

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

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

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

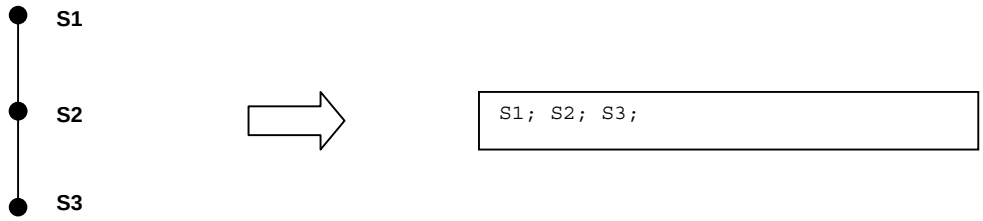


Figure 8: Illustration of sequential behaviour

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

EXAMPLE:

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

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

Table 16 gives an overview of the TTCN 3 expressions, statements and operations and restrictions on their usage.

Table 16: Overview of TTCN-3 expressions, statements and operations

Statement	Associated keyword or symbol	Can be used in module control	Can be used in functions, test cases and altsteps	Can be used in functions called from templates, Boolean guards, or from initialization of altstep local definitions
Expressions	**(…)**	Yes	Yes	Yes
Basic program statements				
Assignments	**:=**	Yes	Yes	Yes (see note 3)
If-else	**if (…) {…} else {…}**	Yes	Yes	Yes
Select case	**select case (…) { case (…) {…} case else {…}}**	Yes	Yes	Yes
For loop	**for (…) {…}**	Yes	Yes	Yes
While loop	**while (…) {…}**	Yes	Yes	Yes
Do while loop	**do {…} while (…)**	Yes	Yes	Yes
Label and Goto	**label / goto**	Yes	Yes	Yes
Stop execution	**stop**	Yes	Yes	
Returning control	**return**		Yes (see note 4)	Yes

Statement	Associated keyword or symbol	Can be used in module control	Can be used in functions, test cases and altsteps	Can be used in functions called from templates, Boolean guards, or from initialization of altstep local definitions
Leaving a loop, alt, altstep or interleave	**break**	Yes	Yes	Yes
Next iteration of a loop	**continue**	Yes	Yes	Yes
Logging	**log**	Yes	Yes	Yes
Statements and operations for alternative behaviours				
Alternative behaviour	**alt {…}**	Yes (see note 1)	Yes	
Re-evaluation of alternative behaviour	**repeat**	Yes (see note 1)	Yes	
Interleaved behaviour	**interleave {…}**	Yes (see note 1)	Yes	
Activate a default	**activate**	Yes (see note 1)	Yes	
Deactivate a default	**deactivate**	Yes (see note 1)	Yes	
Configuration operations				
Create parallel test component	**create**		Yes	
Connect component port to component port	**connect**		Yes	
Disconnect two component ports	**disconnect**		Yes	
Map port to test interface	**map**		Yes	
Unmap port from test system interface	**unmap**		Yes	
Get MTC component reference value	**mtc**		Yes	Yes
Get test system interface component reference value	**system**		Yes	Yes
Get own component reference value	**self**		Yes	Yes
Start execution of test component behaviour	**start**		Yes	
Stop execution of test component behaviour	**stop**		Yes	
Remove a test component from the system	**kill**		Yes	
Check termination of a PTC behaviour	**running**		Yes	
Check if a PTC exists in the test system	**alive**		Yes	
Wait for termination of a PTC behaviour	**done**		Yes	
Wait a PTC cease to exist	**killed**		Yes	
Communication operations				
Send message	**send**		Yes	
Invoke procedure call	**call**		Yes	
Reply to procedure call from remote entity	**reply**		Yes	
Raise exception (to an accepted call)	**raise**		Yes	
Receive message	**receive**		Yes	
Trigger on message	**trigger**		Yes	
Accept procedure call from remote entity	**getcall**		Yes	
Handle response from a previous call	**getreply**		Yes	
Catch exception (from called entity)	**catch**		Yes	
Check (current) message/call received	**check**		Yes	
Clear port queue	**clear**		Yes	
Clear queue and enable sending & receiving at a to port	**start**		Yes	
Disable sending and disallow receiving operations to match at a port	**stop**		Yes	
Disable sending and disallow receiving operations to match new messages/calls	**halt**		Yes	

Statement	Associated keyword or symbol	Can be used in module control	Can be used in functions, test cases and altsteps	Can be used in functions called from templates, Boolean guards, or from initialization of altstep local definitions
Timer operations				
Start timer	**start**	Yes	Yes	
Stop timer	**stop**	Yes	Yes	
Read elapsed time	**read**	Yes	Yes	
Check if timer running	**running**	Yes	Yes	
Timeout event	**timeout**	Yes	Yes	
Verdict operations				
Set local verdict	**setverdict**		Yes	
Get local verdict	**getverdict**		Yes	Yes
External actions				
Stimulate an (SUT) action externally	**action**	Yes	Yes	
Execution of test cases				
Execute test case	**execute**	Yes	Yes (see note 2)	
NOTE 1: Can be used to control timer operations only. NOTE 2: Can only be used in functions and altsteps that are used in module control. NOTE 3: Changing of component variables is disallowed. NOTE 4: Can be used in functions and altsteps but not in test cases.				

19 Basic program statements

The basic program statements presented in table 17 can be used in the control part of a module and in TTCN-3 functions, altsteps and test cases.

Table 17: Overview of TTCN-3 basic program statements

Basic program statements	
Statement	**Associated keyword or symbol**
Assignments	**:=**
If-else	**if (…) {…} else {…}**
Select case	**select case (…) { case (…) {…} case else {…}}**
For loop	**for (…) {…}**
While loop	**while (…) {…}**
Do while loop	**do {…} while (…)**
Label and Goto	**label / goto**
Stop execution	**stop**
Returning control	**return**
Leaving a loop, alt, altstep or interleave	**break**
Next iteration of a loop	**continue**
Logging	**log**

19.1 Assignments

Values or templates may be assigned to variables or template variables (see clause 11). This is indicated by the symbol ":=".

Syntactical Structure

```
VariableRef ":=" ( Expression | TemplateBody )
```

Semantic Description

During execution of an assignment, the right-hand side of the assignment shall evaluate to a value or template. The effect of an assignment is to bind the variable to the value of the expression or to a template. The expression shall contain no unbound variables. Assignments are processed from left to right, i.e. expressions in the left-hand-side are evaluated before those in the right-hand-side. The evaluations obey the operator precedence defined in table 6. The right-hand-side is evaluated completely before the resulting value or template is bound to the evaluated left-hand side of the assignment. Whenever assignments are used within the right-hand-side of an assignment (due to assignment notation), these rules apply recursively.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The right-hand side of an assignment shall evaluate to a value or template, which is type compatible with the variable at the left-hand side of the assignment.

b) When the right-hand side of the assignment evaluates to a template (global or local template, in-line template or template variable), the variable at the left hand side shall be a template variable.

Examples

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

19.2 The If-else statement

The **if-else** statement, also known as the conditional statement, is used to denote branching in the control flow.

Syntactical Structure

```
if "(" BooleanExpression ")" StatementBlock
{ else if "(" BooleanExpression ")" StatementBlock }
[ else StatementBlock]
```

NOTE: **else if** "(" *BooleanExpression* ")" *StatementBlock* [**else** *StatementBlock*] is a shorthand notation for **else** "{" **if** "(" *BooleanExpression* ")" *StatementBlock* [**else** *StatementBlock*] "}".

Semantic Description

The branching of the control flow is decided upon the value of the Boolean expressions - the condition. A statement block - and only one - will be executed, if its condition evaluates to true. The optional else specifies a statement block that will be executed if all the "if" and "else if" conditions before are false.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

```
if (date == "1.1.2005") { return ( fail ); }

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

19.3 The Select case statement

The **select case** statement is an alternative syntactic form of the **if-else** statement.

Syntactical Structure

```
select "(" SingleExpression ")" "{"
     { case "(" { SingleExpression [","] } ")" StatementBlock }
     [ case else StatementBlock ]
"}"
```

Semantic Description

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

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

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

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

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

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The **select** *SingleExpression* and the **case** *SingleExpression*-s shall be type compatible.

Examples

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

// the above select statement is equivalent to the following nested if-else statement.
// Note: the following textual replacement of the select-case statement is described in
// the operational semantics of TTCN-3.
{
    var charstring myTempVar := MyModulePar;
    if (match(myTempVar, "firstValue")
        {
         log ("The first branch is selected");
        }
    else if (match(myTempVar, MyCharVar) or match(myTempVar, MyCharConst))
        {
         log ("The second branch is selected");
        }
```

```
    else
        {
         log ("The value of the module parameter MyModulePar is selected");
        }
}
```

19.4 The For statement

The **for** statement defines a counter loop.

Syntactical Structure

```
for "(" ( VarInstance | Assignment ) ";" BooleanExpression ";" Assignment ")"
        StatementBlock
```

Semantic Description

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

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

The termination criterion of the loop shall be expressed by a **boolean** expression. It is checked at the beginning of each new loop iteration. If it evaluates to **true**, the execution continues with the statement block in the for statement, if it evaluates to **false**, the execution continues with the statement which immediately follows the **for** loop. If a **break** statement is executed that is not within the body of an enclosed loop, **alt**, alststep or **interleave**, then the loop is terminated, too.

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

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

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


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

19.5 The While statement

A **while** statement defines a loop that is executed as long as the loop condition holds.

Syntactical Structure

```
while "(" BooleanExpression ")" StatementBlock
```

Semantic Description

The loop condition shall be checked at the beginning of each new loop iteration. If the loop condition does not hold, then the loop is exited and execution shall continue with the statement, which immediately follows the **while** loop. If a **break** statement is executed that is not within the body of an enclosed loop, **alt**, alststep or **interleave**, then the loop is terminated, too.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

```
while (j<10){ … }
```

19.6 The Do-while statement

A **do-while** statement defines a loop that is executed up until the loop condition does not hold.

Syntactical Structure

do *StatementBlock* **while** "(" *BooleanExpression* ")"

Semantic Description

The **do-while** loop is identical to a **while** loop with the exception that the loop condition shall be checked at the *end* of each loop iteration. This means when using a **do-while** loop the behaviour is executed at least once before the loop condition is evaluated for the first time. If a **break** statement is executed that is not within the body of an enclosed loop, **alt**, alststep or **interleave**, then the loop is terminated, too.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

```
do { … } while (j<10);
```

19.7 The Label statement

The **label** statement allows the specification of labels in test cases, functions, altsteps and the control part of a module.

Syntactical Structure

label *LabelIdentifier*

Semantic Description

A **label** marks a statement. The label is used by the **goto** statement (see clause 19.8) to transfer control to a labelled statement.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) A **label** statement can be used freely like other TTCN-3 behavioural program statements according to the syntax rules defined in annex A. It can be used before or after a TTCN-3 statement but not as the first statement of an alternative or top alternative in an **alt** statement, **interleave** statement or **altstep**.

b) Labels used following the **label** keyword shall be unique among all labels defined in the same test case, function, altstep or control part.

Examples

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

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

```
        PCO2.send(MySig6);
        label L3;                    // Definition of label L3
        PCO2.receive(MySig7);
    }
}
 :
```

19.8 The Goto statement

A **goto** statement performs a jump to a **label.**

Syntactical Structure

goto *LabelIdentifier*

Semantic Description

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

The **goto** statement provides the possibility to jump freely, i.e. forwards and backwards, within a sequence of statements, to jump out of a single compound statement (e.g. a **while** loop) and to jump over several levels out of nested compound statements (e.g. nested alternatives).

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) It is not allowed to jump out of or into functions, test cases, altsteps and the control part of a TTCN-3 module.

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

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

Examples

```
// The following TTCN-3 code fragment includes
:
label L1;                           // … the definition of label L1,
MyVar := 2 * MyVar;
if (MyVar < 2000) { goto L1; }      // … a jump backward to L1,
MyVar2 := Myfunction(MyVar);
if (MyVar2 > MyVar) { goto L2; }    // … a jump forward to L2,
PCO1.send(MyVar);
PCO1.receive;
label L2;                           // … the definition of label L2,
PCO2.send(integer: 21);
alt {
  [] PCO1.receive { }
  [] PCO2.receive(integer: 67) {
        label L3;                   // … the definition of label L3,
        PCO2.send(MyVar);
        alt {
          [] PCO1.receive { }
          [] PCO2.receive(integer: 90) {
                PCO2.send(integer: 33);
                PCO2.receive(integer: 13);
                goto L4;            // … a jump forward out of two nested alt statements,
             }
          [] PCO2.receive(MyError) {
                goto L3;            // … a jump backward out of the current alt statement,
             }
          [] any port.receive {
                goto L2;            // … a jump backward out of two nested alt statements,
             }
        }
     }
  [] any port.receive {
        goto L2;                    // … and a long jump backward out of an alt statement.
     }
}
```

```
label L4;
:
```

19.9 The Stop execution statement

The **stop** statement terminates execution of test components, a test case or a test control.

Syntactical Structure

```
stop
```

Semantic Description

The **stop** statement terminates execution in different ways depending on the context in which it is used. When used in the control part of a module or in a function used by the control part of a module, it terminates the execution of the module control part. When used in a test case, altstep or function that are executed on a test component, it terminates the relevant test component.

NOTE: The semantics of a **stop** statement that terminates a test component is identical to the stop component operation **self.stop** (see clause 21.3.3).

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

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

19.10 The Return statement

The **return** statement terminates execution of functions or altsteps.

Syntactical Structure

```
return [ Expression ]
```

Semantic Description

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

TTCN-3 allows optional statement blocks that may follow altstep calls within **alt** statements. If there is a statement block, the **return** statement returns control to the beginning of this statement block and the statement block is executed before the **alt** statement is left. If there is no statement block, test execution continues with the first statement following the **alt** statement.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The return statement shall not be used in the statement block of a testcase.

Examples

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

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

19.11 The Log statement

The **log** statement provides the means to write logging information to some logging device. The information that can be logged is summarized in table 18.

Table 18: TTCN-3 language elements that can be logged

Used in a log statement	What is logged	Comment
module parameter identifier	actual value	
literal value	value	This includes also free text.
data constant identifier	actual value	
template instance	actual template or field values and matching symbols	
data type variable identifier	actual value or "UNINITIALIZED"	See notes 3 and 4.
self, **mtc**, **system** or component type variable identifier	actual value and if assigned the component instance name or "UNINITIALIZED"	On logging actual values see notes 2 to 4. Actual component states shall be logged according to note 5.
running operation (component or timer)	return value	**true** or **false**. In case of component or timer arrays, array element specification shall be included.
alive operation (component)	return value	**true** or **false**. In case of arrays, array element specifications shall be included.
port instance	actual state	Port states shall be logged according to note 6.
default type variable identifier	actual state or "UNINITIALIZED"	Default states shall be logged according to note 7. See also notes 2 to 4.
timer name	actual state	Timer states shall be logged according to note 8.
read operation	return value	See clause 24.3.
match operation	return value	
getverdict operation	return value	**none**, **pass**, **inconc**, or **fail**
predefined functions	return value	See annex C.
function instance	return value	Only functions with return clause are allowed.
external function instance	return value	Only external functions with return clause are allowed.

Used in a log statement	What is logged	Comment
formal parameter identifier	see comment column	Logging of actual parameters shall follow rules specified for the language elements they are substituting. In case of value parameters the actual parameter value, in case of template-type parameters the actual template or field values and matching symbols, in case of component type parameters the actual component reference etc. shall be logged. For timer parameters also the use of the read operation and for component type and timer parameters the use of the running operation are allowed.
NOTE 1: Actual value/actual template is the value/template at the moment of the execution of the log statement. NOTE 2: The type of the logged value is tool dependent. NOTE 3: In case of array identifiers without array element specification, actual values and for component references names of all array elements shall be logged. NOTE 4: The string "UNINITIALIZED" is logged only if the log item is unbound (uninitialized). NOTE 5: Component states that can be logged are: Inactive, Running, Stopped and Killed (for further details see annex F). NOTE 6: Port states that can be logged are: Started and Stopped (for further details see annex F). NOTE 7: Default states that can be logged are: Activated and Deactivated. NOTE 8: Timer states that can be logged are: Inactive, Running and Expired (for further details see annex F).		

Syntactical Structure

```
log "(" { ( FreeText | TemplateInstance ) [","] } ")"
```

Semantic Description

The **log** statement provides the means to write one or more log items to some logging device associated with the test control or the test component in which the statement is used. Items to be logged shall be identified by a comma-separated list in the argument of the log statement. Log items may be individual language elements specified in table 18 or expressions composed of such log items.

It is strongly recommended that the execution of the **log** statement has no effect on the test behaviour. In particular, functions used in a log statement should not (explicitly or implicitly) change component variable values, port or timer status, and should not change the value of any of its inout or out parameters.

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

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

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

19.12 The Break statement

A **break** statement causes the exit from a loop, from an altstep or from an **alt** or **interleave** statement.

Syntactical Structure

```
break
```

Semantic Description

On executing a **break** statement the innermost, currently executed loop, **alt** statement or **interleave** statement is left. Execution continues with the statement following the construct which is left. Using **break** outside the body of a loop (**for**, **while**, **do-while**) or an alternative of an **alt** or **interleave** statement shall cause a dynamic error.

Altsteps are always executed within a surrounding **alt** statement. If the execution of a top alternative of an altstep (see clause 16.2) ends with a **break** statement, the altstep and the surrounding **alt** statement are left. Execution continues with the statement following the surrounding **alt** statement.

NOTE: TTCN-3 allows optional statement blocks that may follow altstep calls within **alt** statements. These statement blocks are not executed when the altstep is left by executing a **break** statement. A **return** statement has to be used, if such an optional statement block has to be executed **(see clause 19.10)**.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

```
do {
    …
    if (cond1) {
        break;          // the do-while loop is left
    }
    …
    for (var integer j:=1; j<=10; j:= j+1) {
        …
        if (cond2) {
            break;          // the for-loop is left but the do-while loop is continued
        }
        …
    }
    …
}
while (j<10);
```

19.13 The Continue statement

A **continue** statement causes the start of the next iteration of a loop.

Syntactical Structure

```
continue
```

Semantic Description

On executing a **continue** statement, the subsequent statements of the body of the innermost, currently executed loop are skipped and the next iteration starts. Using **continue** outside the body of a loop (**for**, **while**, **do-while**) shall cause a dynamic error.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

```
do {
    …
    if (cond) {
        continue;       // execution continues with the next iteration of the do-while-loop
    }
    …
    …
    for (var integer j:=1; j<=10; j:= j+1) {
        …
        if (cond2) {
            continue;   // continues with the next iteration of the for-loop
        }
```

```
        …
    }
    …
}
while (j<10);
```

19.14 Statement block

Statement blocks can be used like basic program statements to introduce a local scope in the flow of control of TTCN-3 behaviour. The declarations and statements in a statement block are executed in the order of their appearance, i.e. sequentially.

Syntactical Structure

```
"{" { LocalDefinition | Statement } "}"
```

Semantic Description

A statement block defines a local scope unit. Scoping rules for TTCN-3 are defined in clause 5.2.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

```
var integer aVar:= 0;           // aVar is declared

{                               // start of a statement block
    var integer myVar:= 2;      // myVar is declared
    aVar := 5 + myVar;          // myVar is used in an assignment
}                               // end of statement block

    // after leaving the statement block aVar is still known, but myVar is not known anymore.
```

20 Statement and operations for alternative behaviours

Test behaviour cannot only be expressed sequentially, but also as a set of alternatives or combinations of both. An interleaving operator allows the specification of interleaved sequences or alternatives. Table 19 summarizes the statements and operations for alternative behaviours.

Table 19: Overview of TTCN-3 statements and operations for alternative behaviours

Statements and operations for alternative behaviours	
Statement/Operation	**Associated keyword or symbol**
Alternative behaviour	**alt { … }**
Re-evaluation of alt statements	**repeat**
Interleaved behaviour	**interleave { … }**
Activate a default	**activate**
Deactivate a default	**deactivate**

20.1 The snapshot mechanism

A more complex form of behaviour is where sequences of statements are expressed as sets of possible alternatives to form a tree of execution paths, as illustrated in figure 9.

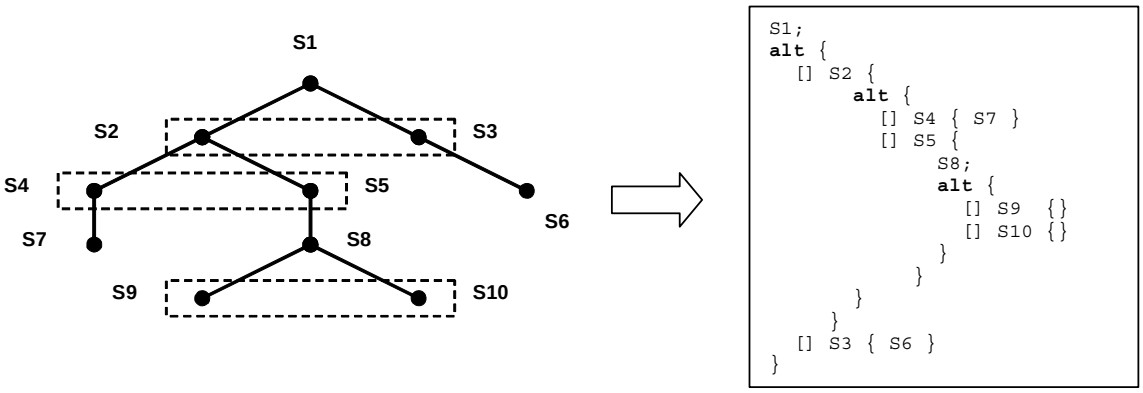


Figure 9: Illustration of alternative behaviour

This is done with the **`alt`** statement.

When entering an **`alt`** statement, a snapshot is taken. A snapshot is considered to be a partial state of a test component that includes all information necessary to evaluate the Boolean conditions that guard alternative branches, all relevant stopped test components, all relevant timeout events and the top messages, calls, replies and exceptions in the relevant incoming port queues. Any test component, timer and port which is referenced in at least one alternative in the **`alt`** statement, or in a top alternative of an altstep that is invoked as an alternative in the **`alt`** statement or activated as default is considered to be relevant. A detailed description of the snapshot semantics is given in the operational semantics of TTCN-3 (part 4 of the TTCN-3 standard - ES 201 873-4 [1]).

NOTE 1: Snapshots are only a conceptual means for describing the behaviour of the **`alt`** statement. The concrete algorithms for the snapshot handling can be found in part 4 of the TTCN-3 standard (ES 201 873-4 [1]).

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

20.2 The Alt statement

An alt statement expresses sets of possible alternatives that form a tree of possible execution paths.

Syntactical Structure

```
alt "{"
          {
            "[" [ BooleanExpression ] "]"
                  ( ( TimeoutStatement |
                       ReceiveStatement |
                       TriggerStatement |
                       GetCallStatement |
                       CatchStatement |
                       CheckStatement |
                       GetReplyStatement |
                       DoneStatement |
                       KilledStatement ) StatementBlock )
                  |
                  ( AltstepInstance [ StatementBlock ] )
          }
          [ "[" else "]" StatementBlock ]
"}"
```

Semantic Description

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

Execution of alternative behaviour:

When entering an **alt** statement, a snapshot is taken.

The alternative branches in the **alt** statement and the top alternatives of invoked altsteps and altsteps that are activated as defaults are processed in the order of their appearance. If several defaults are active, the reverse order of their activation determines the evaluation order of the top alternatives in the defaults. The alternative branches in active defaults are reached by the default mechanism described in clause 20.5.

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

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

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

An *altstep-branch* is selected if the Boolean guard is fulfilled. The selection of an *altstep-branch* causes the invocation of the referenced altstep, i.e. the altstep is invoked and the evaluation of the snapshot continues within the altstep. Altstep-branches may contain an optional statement block. The optional statement block shall be executed only, if an alternative of the altstep referenced in the altstep-branch has been selected and executed.

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

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

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

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

NOTE 1: The TTCN-3 semantics describe the evaluation of a snapshot as a series of indivisible actions of a test component. The semantics do not assume that the evaluation of a snapshot has no duration. During the evaluation of a snapshot, test components may stop, timers may timeout and new messages, calls, replies or exceptions may enter the port queues of the component However, these events do not change the actual snapshot and thus, are not considered for the snapshot evaluation.

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

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

NOTE 2: The repetitive procedure of taking a complete snapshot and re-evaluate all alternatives is only a conceptual means for describing the semantics of the **alt** statement. The concrete algorithm that implements this semantics is outside the scope of the present document.

Selecting/deselecting an alternative:

If necessary, it is possible to enable/disable an alternative by means of a Boolean expression placed between the ("[…]") brackets of the alternative.

Else branch in alternatives:

Any branch in an **alt** statement can be defined as an else branch by including the **else** keyword between the opening and closing brackets at the beginning of the alternative. The statement block of the else branch is always executed if no other alternative textually preceding the else branch has proceeded.

Default mechanism:

It should be noted that the default mechanism (see clause 20.5) is always invoked at the end of all alternatives. If an **else** branch is defined, the default mechanism will never be called, i.e. active defaults will never be entered.

NOTE 3: It is also possible to use **else** in altsteps.

NOTE 4: It is allowed to use a **repeat** statement within an **else** branch.

NOTE 5: It is allowed to define more than one else branch in an alt statement or in an altstep, however always only the first else branch is executed.

Re-evaluation of alt statements:

The re-evaluation of an **alt** statement can be specified by using a **repeat** statement (see clause 20.3).

Invocation of altsteps as alternatives:

TTCN-3 allows the invocation of altsteps as alternatives in **alt** statements (see clause 16.2.1). When an altstep is explicitly invoked as an alternative, the optional statement block following the altstep call shall also be executed.

Continue execution after the alt statement:

Behaviour execution continues with the statement following the **alt** statement when one of the branches of the **alt** or invoked defaults is selected and completely executed, or a branch of an **altstep** used in an altsteps-branch is selected and the branch and the optional statement block following the invoked altstep are completely executed.

Execution also continues with the statement following the **alt** statement if a **break** statement is reached in the statement block of the selected branch of an **alt** statement, of an **altstep** used in an altstep-branch, or of an **altstep** invoked as default.

The **alt** statement can also be left by using a **goto** statement in the selected branch of the **alt** (i.e. no branches of altsteps and defaults can be considered in this case), and execution continues with the statement following the label, **goto** is pointing to.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

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

b) The evaluation of a Boolean expression guarding an alternative may have side-effects. To avoid side effects that cause an inconsistency between the actual snapshot and the state of the component, the same restrictions as the restrictions for the initialization of local definitions within altsteps shall apply (clause 16.2).

c) The else branch shall not contain any of the actions allowed in branches guarded by a boolean expression (i.e. an **altstep** call or a **done**, a **killed**, a **timeout** or a receiving operation).

d) An **alt** statement used within the module control part shall only contain the **timeout** statements.

Examples

EXAMPLE 1: Nested alternatives

```
alt {
    [] MyPort.receive (MyMessage) {
        setverdict (pass);
        MyTimer.start;
        alt {
            [] MyPort.receive (MySecondMessage) {
                MyTimer.stop;
                setverdict (pass);
            }
            [] MyTimer.timeout {
                MyPort.send (MyRepeat);
                MyTimer.start;
                alt {
                    [] MyPort.receive (MySecondMessage) {
                        MyTimer.stop;
                        setverdict (pass)
                    }
                    [] MyTimer.timeout { setverdict (inconc) }
                    [] MyPort.receive { setverdict (fail) }
                }
            }
            [] MyPort.receive { setverdict (fail) }
        }
    }
    [] MyTimer.timeout { setverdict (inconc) }
    [] MyPort.receive { setverdict (fail) }
}
```

EXAMPLE 2: Alt statement with guards

```
alt {
  [x>1] L2.receive {                    // Boolean guard/expression
        setverdict(pass);
     }
  [x<=1] L2.receive {                   // Boolean guard/expression
        setverdict(inconc);
     }
}
```

EXAMPLE 3: Alt statement with else branch

```
// Use of alternative with Boolean expressions (or guard) and else branch
alt {
  :
  [else] {                       // else branch
        MyErrorHandling();
        setverdict(fail);
        stop;
     }
}
```

EXAMPLE 4: Re-evaluation with repeat

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

EXAMPLE 5: Alt statement with explicitly invoked altstep

```
alt {
  [] PCO3.receive { }
  [] AnotherAltStep() {     // Explicit call of altstep AnotherAltStep as alternative.
       setverdict(inconc)   // Statement block executed if an alternative within
                            // altstep AnotherAltStep has been selected and executed.
     }
  [] MyTimer.timeout { }
}
```

20.3 The Repeat statement

The **repeat** statement is used for a re-evaluation of an **alt** statement.

Syntactical Structure

```
repeat
```

Semantic Description

The **repeat** statement, when used in the statement block of alternatives of **alt** statements, causes the re-evaluation of the **alt** statement, i.e. a new snapshot is taken and the alternatives of the **alt** statement are evaluated in the order of their specification.

When used in statement blocks of the response and exception handling parts of blocking procedure calls, the repeat statement causes the re-evaluation of the response and exception handling part of the call (see clause 22.3.1).

If a **repeat** statement is used in a top alternative in an altstep definition, it causes a new snapshot and the re-evaluation of the **alt** statement from which the altstep has been called. The call of the altstep may either be done implicitly by the default mechanism (see clause 20.5.1) or explicitly in the **alt** statement (see clause 20.2).

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The **repeat** statement shall only be used within **alt** statements, **call** statements or altsteps.

Examples

EXAMPLE 1: Usage of repeat in an alt statement

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

EXAMPLE 2: Usage of repeat in an altstep

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

20.4 The Interleave statement

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

Syntactical Structure

```
interleave "{"
     { "[]" ( TimeoutStatement |
              ReceiveStatement |
              TriggerStatement |
              GetCallStatement |
              CatchStatement |
              CheckStatement |
              GetReplyStatement |
              DoneStatement |
              KilledStatement ) StatementBlock
     }
"}"
```

Semantic Description

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

Interleaved behaviour can always be replaced by an equivalent set of nested **alt** statements. The procedures for this replacement and the operational semantics of interleaving are described in part 4 of the TTCN-3 standard (ES 201 873-4 [1]).

The rules for the evaluation of an interleaving statement are the following:

a) Whenever a reception statement is executed, the following non-reception statements are subsequently executed until the next reception statement is reached, a **break** statement is reached, or the interleaved sequence ends.

NOTE 1: Reception statements are TTCN-3 statements which may occur in sets of alternatives, i.e. **receive**, **check**, **trigger**, **getcall**, **getreply**, **catch**, **done, killed** and **timeout**. Non-reception statements denote all other non-control-transfer statements which can be used within the **interleave** statement.

b) If none of the alternatives of the **interleave** statement can be executed, the default mechanism will be invoked. This means, according to the semantics of the default mechanism, the actual snapshot will be used to evaluate those altsteps that have been activated before entering the **interleave** statement.

NOTE 2: The complete semantics of the default mechanism within an **interleave** statement is given by replacing the **interleave** statement by an equivalent set of nested **alt** statements. The default mechanism applies for each of these **alt** statements.

c) The evaluation then continues by taking the next snapshot if no **break** statement was encountered.

d) The evaluation of the **interleave** statement is terminated if a **break** statement is executed.

The operational semantics of interleaving are fully defined in part 4 of the TTCN-3 standard (ES 201 873-4 [1]).

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) Control transfer statements **for**, **while**, **do-while**, **goto**, **activate**, **deactivate**, **stop**, **repeat**, **return**, direct call of altsteps as alternatives and (direct and indirect) calls of user-defined functions, which include communication operations, shall not be used in **interleave** statements.

b) In addition, it is not allowed to guard branches of an **interleave** statement with Boolean expressions (i.e. the '[ ]' shall always be empty). It is also not allowed to specify **else** branches in interleaved behaviour.

Examples

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

// is a shorthand for
alt {
[]  PCO1.receive(MySig1)
    {   PCO1.send(MySig2);
        alt {
        []  PCO1.receive(MySig3)
            {   PCO2.receive(MySig4);
                PCO2.send(MySig5);
                PCO2.send(MySig6);
                PCO2.receive(MySig7)
            }
        []  PCO2.receive(MySig4)
            {   PCO2.send(MySig5);
                PCO2.send(MySig6);
                alt {
                []  PCO1.receive(MySig3) {
                        PCO2.receive(MySig7); }
                []  PCO2.receive(MySig7) {
                        PCO1.receive(MySig3); }
                }
            }
        }
    }
[]  PCO2.receive(MySig4)
    {   PCO2.send(MySig5);
        PCO2.send(MySig6);
        alt {
        []  PCO1.receive(MySig1)
            {   PCO1.send(MySig2);
                alt {
                []  PCO1.receive(MySig3)
                    {   PCO2.receive(MySig7);
                    }
                []  PCO2.receive(MySig7)
                    {   PCO1.receive(MySig3);
                    }
                }
            }
        []  PCO2.receive(MySig7)
            {   PCO1.receive(MySig1);
                PCO1.send(MySig2);
                PCO1.receive(MySig3);
            }
        }
    }
}
```

20.5 Default Handling

TTCN-3 allows the activation of altsteps (see clause 16.2) as defaults. For each test component the defaults, i.e. activated altsteps, are stored as an ordered list. The defaults are listed in the reversed order of their activation i.e. the last activated default is the first element in the list of active defaults. The TTCN-3 operations **`activate`** (see clause 20.5.2) and **`deactivate`** (see clause 20.5.3) operate on the list of defaults. An **`activate`** puts a new default as the first element into the list and a **`deactivate`** removes a default from the list. A default in the default list can be identified by means of default reference that is generated as a result of the corresponding **`activate`** operation.

20.5.1 The default mechanism

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

NOTE 1: An **interleave** statement is semantically equivalent to a nested set of **alt** statements and the default mechanism also applies to each of these **alt** statements. This means, the default mechanism also applies to **interleave** statements.

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

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

In the case of a successful termination, the default may either stop the test component by means of a **stop** statement, or the main control flow of the test component will continue immediately after the **alt** statement from which the default mechanism was called or the test component will take new snapshot and re-evaluate the **alt** statement. The latter has to be specified by means of a **repeat** statement (see clause 20.3). If the execution of the selected top alternative of the default ends with a **break** statement or without a **repeat** statement the control flow of the test component will continue immediately after the **alt** statement.

NOTE 2: TTCN-3 does not restrict the implementation of the default mechanism. It may for example be implemented in form of a process that is implicitly called at the end of each **alt** statement or in form of a separate thread that is only responsible for the default handling. The only requirement is that defaults are called in the reverse order of their activation when the default mechanism has been invoked.

20.5.2 The Activate operation

The **activate** operation is used to activate altsteps as defaults.

Syntactical Structure

```
activate "(" AltstepRef "(" [ { ActualPar [","] } ] ")" ")"
```

Semantic Description

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

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

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

The actual parameters of a parameterized altstep (see clause 16.2.1) that should be activated as a default, shall be provided in the corresponding **activate** statement. This means the actual parameters are bound to the default at the time of its activation (and not e.g. at the time of its invocation by the default mechanism).

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) All timer instances in the actual parameter list shall be declared as component type local timers (see clause 6.2.10.1).

b) An altstep that is activated as a default shall only have **in** parameters, port parameters, or timer parameters.

Examples

EXAMPLE 1: Activation where the default reference is kept

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

EXAMPLE 2: Simple activation

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

EXAMPLE 3: Activation of a parameterized altstep

```
altstep MyAltStep2 ( integer    par_value1, MyType par_value2,
                     MyPortType par_port,   timer  par_timer )
{
 :
}
function MyFunc () runs on MyCompType
{ :
var default MyDefaultVar := null;

MyDefaultVar := activate(MyAltStep2(5, myVar, myCompPort, myCompTimer);
    // MyAltStep2 is activated as default with the actual parameters 5 and
    // the value of myVar. A change of myVar before a call of MyAltStep2 by
    // the default mechanism will not change the actual parameters of the call.
 :
}
```

20.5.3 The Deactivate operation

The **deactivate** operation is used to deactivate defaults, i.e. previously activated altsteps.

Syntactical Structure

deactivate ["(" *VariableRef* | *FunctionInstance* ")"]

Semantic Description

A **deactivate** operation will remove the referenced default from the list of defaults.

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

A **deactivate** operation without parameter deactivates all defaults of a test component.

Calling a **deactivate** operation with the special value **null** has no effect. Calling a **deactivate** operation with an undefined default reference, e.g. an old reference to a default that has already been deactivated or an uninitialized default reference variable, shall cause a runtime error.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The variable associated with *VariableRef* (being a component type variable, a component type parameter, etc.) or the return type associated with *FunctionInstance* must be of default type.

Examples

```
var default MyDefaultVar := null;
var default MyDefVarTwo := activate(MySecondAltStep());
var default MyDefVarThree := activate(MyThirdAltStep());
 :
MyDefaultVar := activate(MyAltStep());
 :
deactivate(MyDefaultVar); // deactivates MyAltStep
 :
deactivate; // deactivates all other defaults, i.e. in this case MySecondAltStep
            // and MyThirdAltStep
```

21 Configuration Operations

Configuration operations are used to set up and control test components. They are summarized in table 20. These operations shall only be used in TTCN-3 test cases, functions and altsteps (i.e. not in the module control part).

Table 20: Overview of TTCN-3 configuration operations

Operation	Explanation	Syntax Examples
Connection Operations		
connect	Connects the port of one test component to the port of another test component	`connect(ptc1:p1, ptc2:p2);`
disconnect	Disconnects two or more connected ports	`disconnect(ptc1:p1, ptc2:p2);`
map	Maps the port of one test component to the port of the test system interface	`map(ptc1:q, system:sutPort1);`
unmap	Unmaps two or more mapped ports	`unmap(ptc1:q, system:sutPort1);`
Test Component Operations		

create	Creation of a normal or alive test component, the distinction between normal and alive test components is made during creation (MTC behaves as a normal test component)	Non-alive test components:
		`var PTCType c := PTCType.create;`
		Alive test components:
		`var PTCType c := PTCType.create alive;`

start	Starting test behaviour on a test component, starting a behaviour does not affect the status of component variables, timers or ports	`c.start(PTCBehaviour());`
stop	Stopping test behaviour on a test component	`c.stop;`
kill	Causes a test component to cease to exist	`c.kill;`
alive	Returns true if the test component has been created and is ready to execute or is executing already a behaviour; otherwise returns false	`if (c.alive) …`
running	Returns true as long as the test component is executing a behaviour; otherwise returns false	`if (c.running) …`
done	Checks whether the function running on a test component has terminated	`c.done;`
killed	Checks whether a test component has ceased to exist	`c.killed { … }`
Test Case Operations		
stop	Terminates the test case with the test verdict **error**	`testcase.stop ( … );`
Reference Operations		
mtc	Gets the reference to the MTC	`connect(mtc:p, ptc:p);`
system	Gets the reference to the test system interface	`map(c:p, system:sutPort);`
self	Gets the reference to the test component that executes this operation	`self.stop;`

21.1 Connection Operations

The ports of a test component can be connected to other components or to the ports of the test system interface (see figure 10). In the case of connections between two test components, the **connect** operation shall be used. When connecting a test component to a test system interface the **map** operation shall be used. The **connect** operation directly connects one port to another with the **in** side connected to the **out** side and vice versa. The **map** operation on the other hand can be seen purely as a name translation defining how communications streams can be referenced.

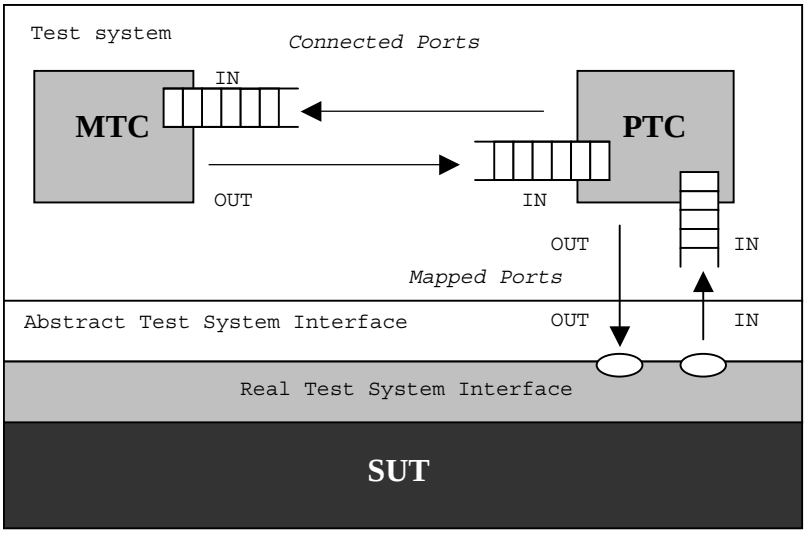


Figure 10: Illustration of the connect and map operations

21.1.1 The Connect and Map operations

The **connect** operation and the **map** operation are used to setup connections to the SUT or between test components.

Syntactical Structure

```
connect "(" ComponentRef ":" Port "," ComponentRef ":" Port ")"

map "(" ComponentRef ":" Port "," ComponentRef ":" Port ")"
     [ param "(" [ { ActualPar [","] }+ ] ")" ]
```

Semantic Description

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

The operation **mtc** identifies the MTC, the operation **system** identifies the test system interface and the operation **self** identifies the test component in which **self** has been called (see clause 6.2.11). All these operations can be used for identifying and connecting ports.

Both the **connect** and **map** operations can be called from any behaviour definition except for the control part of a module. However before either operation is called, the components to be connected shall have been created and their component references shall be known together with the names of the relevant ports.

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

The **map** operation provides an optional parameter list for configuration purposes. The actual parameters must conform to the **map param** clause of the port type declaration of the system port used. It allows to pass values needed for dynamic runtime configuration.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) For both the **`connect`** and **`map`** operations, only consistent connections are allowed.

   Assuming the following:

   1) ports PORT1 and PORT2 are the ports to be connected;
   2) inlist-PORT1 defines the messages or procedures of the in-direction of PORT1;
   3) outlist-PORT1defines the messages or procedures of the out-direction of PORT1;
   4) inlist-PORT2 defines the messages or procedures of the in-direction of PORT2; and
   5) outlist-PORT2 defines the messages or procedures of the out-direction of PORT2.

b) The **`connect`** operation is allowed if and only if:

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

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

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

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

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

f) In addition, the restrictions on allowed and disallowed connections described in clause 9.1 apply.

Examples

EXAMPLE 1: Simple map and connect

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

EXAMPLE 2: Parameterized map

```
 :
var MyConfigType MyConfig := { option := 1, lock := false};
 :
map(mtc:Port4, system:PCO2) param (MyConfig);
 :
// In this example by means of the map operation, Port4 of the MTC is connected to the port PCO2
// of the test system interface, and additionally a parameter containing configuration options
// for the connection is passed.
```

21.1.2 The Disconnect and Unmap operations

The **disconnect** and **unmap** operations are the opposite operations of **connect** and **map**.

Syntactical Structure

```
disconnect [ ( "(" ComponentRef ":" Port "," ComponentRef ":" Port ")" ) |
             ( "(" PortRef ")" ) |
             ( "(" ComponentRef ":" all port ")" ) |
             ( "(" all component ":" all port ")" ) ]

unmap [ ( "(" ComponentRef ":" Port "," ComponentRef ":" Port ")" ) |
             ( "(" PortRef ")" ) |
             ( "(" ComponentRef ":" all port ")" ) |
             ( "(" all component ":" all port ")" ) ]
        [ param "(" [ { ActualPar ["," ] }+ ] ")" ]
```

Semantic Description

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

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

To ease **disconnect** and **unmap** operations related to all connections and mappings of a component or a port, it is allowed to use **disconnect** and **unmap** operations with one argument only. This one argument specifies one side of the connections to be disconnected or unmapped. The **all port** keyword can be used to denote all ports of a component.

The usage of a **disconnect** or **unmap** operation without any parameters is a shorthand form for using the operation with the parameter **self**:**all port**. It disconnects or unmaps all ports of the component that calls the operation.

The **all component** keyword shall only be used in combination with the **all port** keyword, i.e. **all component**:**all port**, and shall only be used by the MTC. Furthermore, the **all component**:**all port** argument shall be used as the one and only argument of a **disconnect** or **unmap** operation and it allows to release all connections and mappings of the test configuration.

Similar to the **map** operation, **unmap** provides an optional parameter list for configuration purposes. The actual parameters must conform to the **unmap param** clause of the port type declaration of the system port used. It allows to pass values needed for dynamic runtime configuration.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

EXAMPLE 1: Disconnect/unmap for specific connections

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

EXAMPLE 2: Disconnect/unmap for a component

```
disconnect(MyNewComponent:Port1);               // disconnects all connections of Port1, which
                                                // is owned by component MyNewComponent.
unmap(MyNewComponent:all port);                 // unmaps all ports of component MyNewComponent
```

EXAMPLE 3: Disconnect/unmap for "self"

```
disconnect;                                     // is a shorthand form for …
disconnect(self:all port);                      // which disconnects all ports of the component
                                                // that called the operation
 :
```

```
unmap;                                          // is a shorthand form for …
unmap(self:all port);                           // which unmaps all ports of the component
                                                // that called the operation
```

EXAMPLE 4: Disconnect/unmap for "all component"

```
disconnect(all component:all port);             // the MTC disconnects all ports of all
                                                // components in the test configuration.
 :
unmap(all component:all port);                  // the MTC unmaps all ports of all
                                                // components in the test configuration.
```

21.2 Test case operations

Test case operations address the entire test case by using the keyword testcase. Currently, the test case stop operation is the only test case operation. It specifies an immediate stop of the test case behaviour with an error verdict.

21.2.1 Test case stop operation

The testcase stop operation defines a user defined immediate termination of a test case with the test verdict **error** and an (optional) associated reason for the termination. Such an immediate stop of a test case is required for cases where a user defined behaviour that does not contribute to the test outcome behaves in an unexpected manner which leads to a situation where the continuation of the test case makes no more sense.

Syntactical Structure

```
testcase "." stop [ "(" { ( FreeText | TemplateInstance ) [","] } ")" ]
```

Semantic Description

The test case stop operation causes an immediate stop of the entire test case behaviour with the verdict **error**. In addition, the test case stop operation provides the means to specify the reason for the immediate termination of a test case by writing one or more items to some logging device associated with the test control or the test component in which the operation is used. Items to be logged shall be identified by a comma-separated list in the argument of the test case stop operation. The argument of the test case stop operation shall follow the same restrictions as the argument of the log statement (see clause 19.11).

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The test case stop operation shall not be used in the module control part or functions invoked directly or indirectly by the module control part.

Examples

```
testcase.stop("Unexpected Termination");
// The test case stops the an error verdict and the string "Unexpected Termination"
// is written to some log device of the test system
```

21.3 Test Component Operations

Test component operations are used to create, start, stop and kill test components. They can also be used to check if test components are alive, running, done or killed.

21.3.1 The Create operation

The **create** operation is used to create test components.

Syntactical Structure

```
ComponentType "." create [ "(" Expression ["," Expression] ")" ] [ alive ]
```

Semantic Description

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

All component variables and timers are reset to their initial value (if any) and all component constants are reset to their assigned values when the component is explicitly or implicitly created.

Two types of PTCs are distinguished: a PTC that can execute a behaviour function only once and a PTC that is kept alive after termination of a behaviour function and can be therefore reused to execute another function. The latter is created using the additional **alive** keyword. An alive-type PTC must be destroyed explicitly using the **kill** operation (see clause 21.3.4), whereas a non-alive PTC is destroyed implicitly after its behaviour function terminates. Termination of a test case, i.e. the MTC, terminates all PTCs that still exist, if any.

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

The **create** operation shall return the unique component reference of the newly created instance. The unique reference to the component will typically be stored in a variable (see clause 6.2.10.1) and can be used for connecting instances and for communication purposes such as sending and receiving.

Optionally, a name can be associated with the newly created component instance. The test system shall associate the names 'MTC' to the MTC and 'SYSTEM' to the test system interface automatically at creation. Associated component names are not required to be unique.

The component instance name is used for logging purposes (see clause 19.11) only and shall not be used to refer to the component instance (the component reference shall be used for this purpose) and has no effect on matching.

Also optionally, a host id can be associated with the newly created component instance. If a host id is provided, the **create** operation shall cause a test case error, if the component cannot be deployed on the specified host.

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

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The name given by the first *Expression* shall be a **charstring** value and when assigned it shall appear as the first argument of the **create** function.

b) The host id given by the second *Expression* shall be a **charstring** value and, when assigned, it shall appear as the second argument of the **create** function.

Examples

```
// This example declares variables of type MyComponentType, which is used to store the
// references of newly created component instances of type MyComponentType which is the
// result of the create operations. An associated name is allocated to some of the created
// component instances.
 :
var MyComponentType MyNewComponent;
var MyComponentType MyNewestComponent;
var MyComponentType MyAliveComponent;
var MyComponentType MyAnotherAliveComponent;
var MyComponentType MyDeployedComponent;
 :
MyNewComponent := MyComponentType.create;
MyNewestComponent := MyComponentType.create("Newest");
MyAliveComponent := MyComponentType.create alive;
MyAnotherAliveComponent := MyComponentType.create("Another Alive") alive;
MyDeployedComponent := MyComponentType.create(-, "Host4");
```

21.3.2 The Start test component operation

The start operation is used to associate a test behaviour to a test component, which is then being executed by that test component.

Syntactical Structure

```
( VariableRef | FunctionInstance ) "." start "(" FunctionInstance ")"
```

Semantic Description

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

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

An alive-type PTC may perform several behaviour functions in sequential order. Starting a second behaviour function on a non-alive PTC or starting a function on a PTC that is still running results in a test case error. If a function is started on an alive-type PTC after termination of a previous function, it uses variable values, timers, ports, and the local verdict as they were left after termination of the previous function. In particular, if a timer was started in the previous function, the subsequent function should be enabled to handle a possible timeout event. In contrast to that, all active defaults are deactivated when the behaviour of an alive-type PTC is stopped. This means no default is activated when a new behaviour is started on an alive-type PTC.

NOTE 1: The lifetime of variables and timers is bound to the scope in which they are declared. When an alive-type component is stopped, only the component scope is left. This means only variable values and timers declared in the component type definition of an alive-type PTC can be accessed by a function with a corresponding **runs on**-clause that is started on an alive-type PTC.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The variable associated with *VariableRef* (being a component type variable, a component type parameter, etc.) or the return type associated with *FunctionInstance* must be of component type.

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

- This function shall have a **runs on** definition referencing a component type that is compatible with the newly created component (see clause 6.3.3).
- Ports and timers shall not be passed into this function.

NOTE 2: Possible return values of a function invoked in a **start** test component operation, i.e. templates denoted by **return** keyword or **inout** and **out** parameters, have no effect when the started test component terminates.

NOTE 3: As **in** and **inout** ports starts listening when the component is created, at the moment, when it starts execution there may be messages in the incoming queues of such ports already waiting to be processed.

Examples

```
function MyFirstBehaviour() runs on MyComponentType { … }
function MySecondBehaviour() runs on MyComponentType { … }
:
var MyComponentType MyNewPTC;
var MyComponentType MyAlivePTC;
:
MyNewPTC := MyComponentType.create;          // Creation of a new non-alive test component.
MyAlivePTC := MyComponentType.create alive; // Creation of a new alive-type test component
:
MyNewPTC.start(MyFirstBehaviour());     // Start of the non-alive component.
MyNewPTC.done;                              // Wait for termination
MyNewPTC.start(MySecondBehaviour());        // Test case error
```

```
:
MyAlivePTC.start(MyFirstBehaviour());       // Start of the alive-type component
MyAlivePTC.done;                            // Wait for termination
MyAlivePTC.start(MySecondBehaviour());      // Start of the next function on the same component
```

21.3.3 The Stop test behaviour operation

The stop test behaviour operation is used to stop the execution of a test component by itself or by another test component.

Syntactical Structure

```
stop |
( ( VariableRef | FunctionInstance | mtc | self ) "." stop ) |
( all component "." stop )
```

Semantic Description

By using the **stop** test component statement a test component can stop the execution of its own currently running test behaviour or the execution of the test behaviour running on another test component. If a component does not stop its own behaviour, but the behaviour running on another test component in the test system, the component to be stopped has to be identified by using its component reference. A component can stop its own behaviour by using a simple **stop** execution statement (see clause 19.9) or by addressing itself in the **stop** operation, e.g. by using the **self** operation.

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

Stopping a test component is the explicit form of terminating the execution of the currently running behaviour. A test component behaviour terminates also by completing its execution upon reaching the end of the testcase or function that is started on this component or by an explicit **return** statement. This termination is also called implicit stop. The implicit stop has the same effects as an explicit stop, i.e. the global verdict is updated with the local verdict of the stopped test component (see clause 24).

If the stopped test component is the MTC, resources of all existing PTCs shall be released, the PTCs shall be removed from the test system and the test case shall terminate (see clause 26.1).

Stopping a non-alive-type test component (implicitly or explicitly) shall destroy it and all resources associated with the test component shall be released.

Stopping an alive-type component shall stop the currently running behaviour only but the component continues to exist and can execute new behaviour (started on it using the **start** operation). Stopping an alive-type component means that all variables, timers and ports declared in the component type definition of the alive-type component keep their value, contents or state. Furthermore, the local verdict of the component keeps its value. In contrast to that, all active defaults are automatically deactivated when the alive-type component is stopped. The component shall be left in a consistent state after stopping its behaviour.

For example, if the behaviour of an alive-type component is stopped during assigning a new value to an already bound variable, the variable shall remain bound after the component is stopped (with the old or the new value). Similarly, if the component is stopped during re-starting an already running timer, the timer shall be left in the running state after termination of the behaviour.

The **all** keyword can be used by the MTC only in order to stop all running PTCs but the MTC itself.

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

NOTE 3: The concrete mechanism for stopping PTCs is outside the scope of the present document.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The variable associated with *VariableRef* (being a component type variable, a component type parameter, etc.) or the return type associated with *FunctionInstance* must be of component type.

Examples

EXAMPLE 1: Stopping another test component and a test component by itself

```
var MyComponentType MyComp := MyComponentType.create;   // A new test component is created
MyComp.start(CompBehaviour());                          // The new component is started
:
if (date == "1.1.2005") {
    MyComp.stop;                    // The component "MyComp" is stopped
}

:
if (a < b ) {
     :
    self.stop;      // The test component that is currently executing stops its own behaviour
}
:
stop            // The test component stops its own behaviour
```

EXAMPLE 2: Stopping all PTCs by the MTC

```
all component.stop      // The MTC stops all PTCs of the test case but not itself.
```

21.3.4 The Kill test component operation

The **kill** test component operation is used to destroy a test component by itself or by another test component. Kill and stop on a non-alive component have the same results, while they differ for alive components: stopping an alive components stops the test behaviour only, the test component continues to exist. Killing a test component destroys the test component.

Syntactical Structure

```
kill |
( ( VariableRef | FunctionInstance | mtc | self ) "." kill ) |
( all component "." kill )
```

Semantic Description

The **kill** operation applied on a test component stops the execution of the currently running behaviour - if any - of that component and frees all resources associated to it (including all port connections of the killed component) and removes the component from the test system. The **kill** operation can be applied on the current test component itself by a simple **kill** statement or by addressing itself using the **self** operation in conjunction with the kill operation. The **kill** operation can also be applied to another test component. In this case the component to be killed shall be addressed using its component reference. If the **kill** operation is applied on the MTC, e.g. **mtc.kill**, it terminates the test case.

The **all** keyword can be used by the MTC only in order to stop and kill all running PTCs but the MTC itself.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The variable associated with *VariableRef* (being a component type variable, a component type parameter, etc.) or the return type associated with *FunctionInstance* must be of component type.

Examples

EXAMPLE 1: Killing another test component and a test component by itself

```
var PTCType MyAliveComp := PTCType.create alive;    // Create an alive-type test component
MyAliveComp.start(MyFirstBehaviour());              // The new component is started
MyAliveComp.done;                                   // Wait for termination
MyAliveComp.start(MySecondBehavior());              // Start the component a 2nd time
MyAliveComp.done;                                   // Wait for termination
MyAliveComp.kill;                                   // Free its resources
```

EXAMPLE 2: Killing all PTCs by the MTC

```
all component.kill;     // The MTC stops all (alive-type and normal) PTCs of the test case first
// and frees their resources.
```

21.3.5 The Alive operation

The **alive** operation is a Boolean operation that checks whether a test component has been created and is ready to execute or is executing already a behaviour function.

Syntactical Structure

```
( VariableRef |
  FunctionInstance |
  any component |
  all component ) "." alive
```

Semantic Description

Applied on a normal test component, the **alive** operation returns true if the component is inactive or running a function and false otherwise. Applied on an alive-type test component, the operation returns true if the component is inactive, running or stopped. It returns false if the component has been killed.

The **alive** operation can be used similar to the **running** operation on PTCSs only (see clause 21.2.6). In particular, in combination with the **all** keyword it returns true if all (alive-type or normal) PTCs are alive.

The **alive** operation used in combination with the **any** keyword returns true if at least one PTC is alive.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The variable associated with *VariableRef* (being a component type variable, a component type parameter, etc.) or the return type associated with *FunctionInstance* must be of component type.

Examples

```
PTC1.done;                              // Waits for termination of the component
if (PTC1.alive) {                       // If the component is still alive …
    PTC1.start(AnotherFunction());      // … execute another function on it.
}
```

21.3.6 The Running operation

The **running** operation is a Boolean operation that checks whether a test component is executing already a behaviour function.

Syntactical Structure

```
( VariableRef |
  FunctionInstance |
  any component |
  all component ) "." running
```

Semantic Description

The **running** operation allows behaviour executing on a test component to ascertain whether behaviour running on a different test component has completed. The **running** operation can be used for PTCs only. The running operation returns **true** for PTCs that have been started but not yet terminated or stopped. It returns **false** otherwise. The **running** operation is considered to be a **boolean** expression and, thus, returns a **boolean** value to indicate whether the specified test component (or all test components) has terminated. In contrast to the **done** operation, the **running** operation can be used freely in **boolean** expressions.

When the **all** keyword is used with the **running** operation, it will return **true** if all PTCs started but not stopped explicitly by another component are executing their behaviour. Otherwise it returns **false**.

When the **any** keyword is used with the **running** operation, it will return **true** if at least one PTC is executing its behaviour. Otherwise it returns **false**.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The variable associated with *VariableRef* (being a component type variable, a component type parameter, etc.) or the return type associated with *FunctionInstance* must be of component type.

Examples

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

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

21.3.7 The Done operation

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

Syntactical Structure

```
( VariableRef |
  FunctionInstance |
  any component |
  all component ) "." done
```

Semantic Description

The **done** operation shall be used in the same manner as a receiving operation or a **timeout** operation. This means it shall not be used in a **boolean** expression, but it can be used to determine an alternative in an **alt** statement or as stand-alone statement in a behaviour description. In the latter case a **done** operation is considered to be a shorthand for an **alt** statement with the **done** operation as the only alternative.

When the done operation is applied to a PTC, it matches only if the behaviour of that PTC has been stopped (implicitly or explicitly) or the PTC has been killed. Otherwise, the match is unsuccessful.

When the **all** keyword is used with the **done** operation, it matches if no one PTC is executing its behaviour. It also matches if no PTC has been created.

When the **any** keyword is used with the **done** operation, it matches if at least the behaviour of one PTC has been stopped or killed. Otherwise, the match is unsuccessful.

NOTE: Stopping the behaviour of a non-alive component also results in removing that component from the test system, while stopping an alive-type component leaves the component alive in the test system. In both cases the **done** operation matches.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The **done** operation can be used for PTCs only.

b) The variable associated with *VariableRef* (being a component type variable, a component type parameter, etc.) or the return type associated with *FunctionInstance* must be of component type.

Examples

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

    []  any port.receive {
            repeat
        }
}

var MyComp c := MyComp.create alive;
c.start(MyPTCBehaviour());
:
c.done;
    // matches as soon as the function MyPTCBehaviour (or function/altstep called by it) stops
c.done;
    // matches the end of MyPTCBehaviour (or function/altstep called by it) too
if(c.running) {c.done}
    // done here matches the end of the next behaviour only

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

// has the following meaning:
alt {
    []  all component.done {}
}
// and thus, blocks the execution until all parallel test components have terminated
```

21.3.8 The Killed operation

The **killed** operation allows to ascertain whether a different test component is alive or has been removed from the test system.

Syntactical Structure

```
( VariableRef |
  FunctionInstance |
  any component |
  all component ) "." killed
```

Semantic Description

The **killed** operation shall be used in the same manner as receiving operations. This means it shall not be used in **boolean** expressions, but it can be used to determine an alternative in an **alt** statement or as a stand-alone statement in a behaviour description. In the latter case a **killed** operation is considered to be a shorthand for an **alt** statement with the **killed** operation as the only alternative.

NOTE: When checking normal test components a killed operation matches if it stopped (implicitly or explicitly) the execution of its behaviour or has been **killed** explicitly, i.e. the operation is equivalent to the **done** operation (see clause 21.2.7). When checking alive-type test components, however, the **killed** operation matches only if the component has been killed using the **kill** operation. Otherwise the **killed** operation is unsuccessful.

When the **all** keyword is used with the **killed** operation, it matches if all PTCs of the test case have ceased to exist. It also matches if no PTC has been created.

When the **any** keyword is used with the **killed** operation, it matches if at least one PTC ceased to exist. Otherwise, the match is unsuccessful.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The **killed** operation can be used for PTCs only.

Examples

```
var MyPTCType ptc := MyPTCType.create alive;     // create an alive-type test component
timer T:= 10.0;                                  // create a timer
T.start;                                         // start the timer
ptc.start(MyTestBehavior());                     // start executing a function on the PTC
alt {
[] ptc.killed {                                  // if the PTC was killed during execution …
    T.stop;                                      // … stop the timer and …
    setverdict(inconc);                          // … set the verdict to 'inconclusive'
   }
[] ptc.done {                                    // if the PTC terminated regularly …
    T.stop;                                      // … stop the timer and …
    ptc.start(AnotherFunction());                // … start another function on the PTC
   }
[] T.timeout {                                   // if the timeout occurs before the PTC stopped
    ptc.kill;                                    // … kill the PTC and …
    setverdict(fail);                            // … set the verdict to 'fail'
   }
}
```

21.3.9 Summary of the use of any and all with components

The keywords **any** and **all** may be used with configuration operations as indicated in table 21.

Table 21: Any and All with components

Operation	Allowed		Example	Comment
	any (see note)	**all** (see note)		
create				
start				
running	Yes but from MTC only	Yes but from MTC only	**any component.running;** **all component.running;**	Is there any PTC performing test behaviour? Are all PTCs performing test behaviour?
alive	Yes but from MTC only	Yes but from MTC only	**any component.alive;** **all component.alive;**	Is there any alive PTC? Are all PTCs alive?
done	Yes but from MTC only	Yes but from MTC only	**any component.done;** **all component.done;**	Is there any PTC that completed execution? Did all PTCs complete their execution?
killed	Yes but from MTC only	Yes but from MTC only	**any component.killed;** **all component.killed;**	Is there any PTC that ceased to exist? Did all PTCs cease to exist?
stop		Yes but from MTC only	**all component.stop;**	Stop the behaviour on all PTCs.
kill		Yes but from MTC only	**all component.kill;**	Kill all PTCs, i.e. they cease to exist.
NOTE: **any** and **all** refer to PTCs only, i.e. the MTC is not considered.				

22 Communication operations

TTCN-3 supports *message-based* and *procedure-based unicast*, *multicast* and *broadcast* communication. Furthermore, TTCN-3 allows to examine the top element of incoming port queues and to control the access to ports by means of *controlling operations*. The communication operations and restrictions on their usage are summarized in table 22.

Table 22: Overview of TTCN-3 communication operations

Communication operations			
Communication operation	**Keyword**	**Can be used at message-based ports**	**Can be used at procedure-based ports**
Message-based communication			
Send message	**send**	Yes	
Receive message	**receive**	Yes	
Trigger on message	**trigger**	Yes	
Procedure-based communication			
Invoke procedure call	**call**		Yes
Accept procedure call from remote entity	**getcall**		Yes
Reply to procedure call from remote entity	**reply**		Yes
Raise exception (to an accepted call)	**raise**		Yes
Handle response from a previous call	**getreply**		Yes
Catch exception (from called entity)	**catch**		Yes
Examine top element of incoming port queues			
Check msg/call/exception/reply received	**check**	Yes	Yes
Controlling operations			
Clear port queue	**clear**	Yes	Yes
Clear queue and enable sending and receiving at a port	**start**	Yes	Yes
Disable sending and disallow receiving operations to match at a port	**stop**	Yes	Yes
Disable sending and disallow receiving operations to match new messages/calls	**halt**	Yes	Yes

22.1 The communication mechanisms

This clause explains the principles of TTCN-3 communication for message-based communication (see clause 22.1.1), for procedure-based communication (see clause 22.1.2), for unicast, multicast, and broadcast communication (see clause 22.1.3), as well as the general format of sending and receiving operations (see clause 22.1.4).

22.1.1 Principles of message-based communication

Message-based communication is communication based on an asynchronous message exchange. Message-based communication is non-blocking on the **`send`** operation, as illustrated in figure 11, where processing in the SENDER continues immediately after the **`send`** operation occurs. The RECEIVER is blocked on the **`receive`** operation until it processes the received message.

In addition to the **`receive`** operation, TTCN-3 provides a **`trigger`** operation that filters messages with certain matching criteria from a stream of received messages on a given incoming port. Messages at the top of the queue that do not fulfil the matching criteria are removed from the port without any further action.

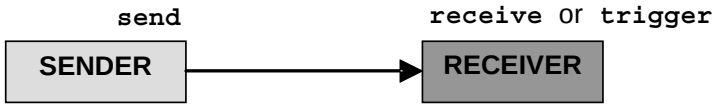


Figure 11: Illustration of the asynchronous send and receive

22.1.2 Principles of procedure-based communication

The principle of procedure-based communication is to call procedures in remote entities. TTCN-3 supports *blocking* and *non-blocking* procedure-based communication. Blocking procedure-based communication is blocking on the calling and the called side, whereas non-blocking procedure-based communication is only blocking on the called side. Signatures of procedures that are used for non-blocking procedure-based communication shall be specified according to the rules in clause 13.

The communication scheme of blocking procedure-based communication is shown in figure 12. The CALLER calls a remote procedure in the CALLEE by using the **call** operation. The CALLEE accepts the call by means of a **getcall** operation and reacts by either using a **reply** operation to answer the call or by raising (**raise** operation) an exception. The CALLER handles the reply or exception by using **getreply** or **catch** operations. In figure 12, the blocking of CALLER and CALLEE is indicated by means of dashed lines.

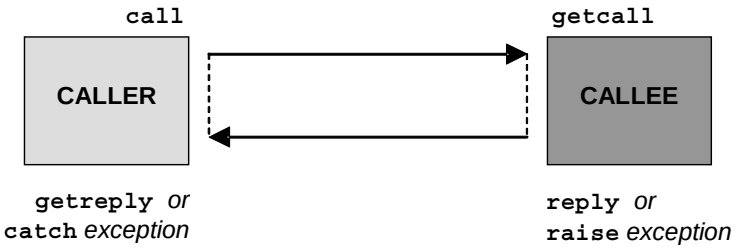


Figure 12: Illustration of blocking procedure-based communication

The communication scheme of non-blocking procedure-based communication is shown in figure 13. The CALLER calls a remote procedure in the CALLEE by using the **call** operation and continues its execution, i.e. does not wait for a reply or exception. The CALLEE accepts the call by means of a **getcall** operation and executes the requested procedure. If the execution is not successful, the CALLEE may raise an exception to inform the CALLER. The CALLER may handle the exception by using a **catch** operation in an **alt** statement. In figure 13, the blocking of the CALLEE until the end of the call handling and possible raise of an exception is indicated by means of a dashed line.

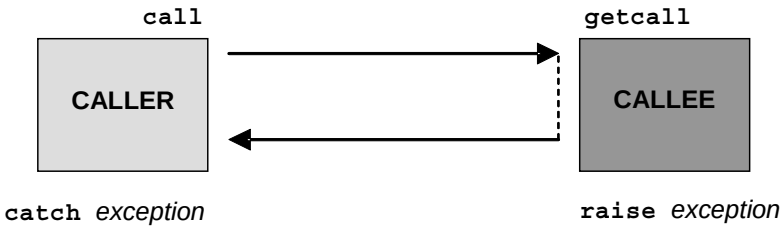


Figure 13: Illustration of non-blocking procedure-based communication

22.1.3 Principles of unicast, multicast and broadcast communication

TTCN-3 supports unicast, multicast and broadcast communication:

- Unicast communication means one sender to one receiver.
- Multicast communication is from one sender to a list of receivers.
- Broadcast communication is from one sender to all receivers (being connected or mapped to the sender).

The terms unicast, multicast and broadcast communication are related to port communication. This means, it is only possible to address one, several or all test components that are connected to the specified port. Unicast, multicast and broadcast can also be used for mapped ports. In this case, one, several or all entities within the SUT can be reached via the specified mapped port.

22.1.4 General format of communication operations

Operations such as **send** and **call** are used for the exchange of information among test components and between an SUT and test components. For explaining the general format of these operations, they can be structured into two groups:

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

b) a component receives a message (**receive** operation), awaits a message (**trigger** operation),accepts a procedure call (**getcall** operation), receives a reply for a previously called procedure (**getreply** operation) or catches an exception (**catch** operation). These actions are collectively referred to as *receiving operations*.

22.1.4.1 General format of the sending operations

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

The send part:

- specifies the port at which the specified operation shall take place;
- defines the message or procedure call to be transmitted;
- gives an (optional) address part that uniquely identifies one or more communication partners to which a message, call, reply or exception shall be send.

The port name, operation name and value shall be present in all sending operations. The address part (denoted by the **to** keyword) is optional and need only be specified in cases of one-to-many connections where:

- unicast communication is used and one receiving entity shall be explicitly identified;
- multicast communication is used and a set of receiving entities has to be explicitly identified;
- broadcast communication is used and all entities connected to the specified port have to be addressed.

EXAMPLE 1:

Send part			(Optional) response and exception
Port and operation	**Value part**	**(Optional) address part**	**handling part**
`MyP1.send`	`(MyVariable + YourVariable - 2)`	`to MyPartner;`	

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

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

EXAMPLE 2:

Send part			(Optional) response and exception handling part
Port and operation	**Value part**	**(Optional) address part**	

`MyP1.call`	`(MyProc:{MyVar1})`		`{`
			`  [] MyP1.getreply(MyProc:{MyVar2}) {}`
			`  [] MyP1.catch(MyProc, ExceptionOne) {}`
			`}`

22.1.4.2 General format of the receiving operations

A receiving operation consists of a *receive* part and an (optional) *assignment* part.

The receive part:

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

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

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

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

The assignment part in a receiving operation is optional. For message-based ports it is used when it is required to store received messages. In the case of procedure-based ports it is used for storing the **in** and **inout** parameters of an accepted call, for storing the return value or for storing exceptions. For the assignment part strong typing is required, e.g. the variable used for storing a message shall have the same type as the incoming message.

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

EXAMPLE:

Receive part				(Optional) assignment part		
Port and operation	**Matching part**	**(Optional) address expression**		**(Optional) value assignment**	**(Optional) parameter value assignment**	**(Optional) sender value assignment**
`MyP1.getreply`	`(AProc:{?} value 5)`		`->`		`param (V1)`	`sender APeer`

Receive part				(Optional) assignment part		
Port and operation	**Matching part**	**(Optional) address expression**		**(Optional) value assignment**	**(Optional) parameter value assignment**	**(Optional) sender value assignment**
`MyP2.receive`	`(MyTemplate(5,7))`	`from APeer`	`->`	`value MyVar`		

22.2 Message-based communication

The operations for message-based communication via asynchronous ports are summarized in table 23.

Table 23: Overview of TTCN-3 message-based communication

Communication operation	Keyword
Send message	**send**
Receive message	**receive**
Trigger on message	**trigger**
Check message received	**check**

22.2.1 The Send operation

The **send** operation is used to place a message on an outgoing message port.

Syntactical Structure

```
Port "." send "(" TemplateInstance ")"
[ to Address ]
```

NOTE: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**all component**".

Semantic Description

The **send** operation places a message on an outgoing message port. The message may be specified by referencing a defined template or can be defined as an in-line template.

Sending unicast, multicast or broadcast

Unicast, multicast and broadcast communication can be determined by the optional **to** clause in the **send** operation. A **to** clause can be omitted in case of a one-to-one connection where unicast communication is used and the message receiver is uniquely determined by the test system structure.

Unicast communication is specified, if the **to** clause addresses one communication partner only. Multicast communication is used, if the **to** clause includes a list of communication partners. Broadcast is defined by using the **to** clause with **all component** keyword.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The TemplateInstance (and all parts of it) shall have a specific value i.e. the use of matching mechanisms such as *AnyValue* is not allowed.

b) When defining the message in-line, the optional type part shall be used if there is ambiguity of the type of the message being sent.

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

d) A **to** clause shall be present in case of one-to-many connections.

e) *AddressRef* shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **send** operation.

Examples

EXAMPLE 1: Simple send (receiver is determined from the test configuration)

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

MyPort.send(5);                     // Sends the integer value 5 (which is an in-line template)
```

EXAMPLE 2: Sending with explicit to clause

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

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

MyPCO2.send(MyTemplate) to (MyPeerOne, MyPeerTwo);
                                  // Specifies a multicast communication, where the value of
                                  // MyTemplate is sent to the two component references stored
                                  // in the variables MyPeerOne and MyPeerTwo.

MyPCO3.send(MyTemplate) to all component;
                                  // Broadcast communication: the value of Mytemplate is send to
                                  // all components which can be addressed via this port. If
                                  // MyPCO3 is a mapped port, the components may reside inside
                                  // the SUT.
```

22.2.2 The Receive operation

The **receive** operation is used to receive a message from an incoming message port queue.

Syntactical Structure

```
( Port | any port ) "." receive
[ "(" TemplateInstance ")" ]
[ from Address ]
[ "->" [ value ( VariableRef |
                 ( "(" { VariableRef [ ":=" FieldOrTypeReference ] [","] } ")" )
               ) ]
       [ sender VariableRef ] ]
```

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**any component**".

Semantic Description

The **receive** operation is used to receive a message from an incoming message port queue. The message may be specified by referencing a defined template or can be defined as an in-line template.

The **receive** operation removes the top message from the associated incoming port queue if, and only if, that top message satisfies all the matching criteria associated with the **receive** operation.

If the match is not successful, the top message shall not be removed from the port queue i.e. if the **receive** operation is used as an alternative of an **alt** statement and it is not successful, the execution of the test case shall continue with the next alternative of the **alt** statement.

Matching criteria

The matching criteria are related to the type and value of the message to be received. The type and value of the message to be received are determined by the argument of the **receive** operation, i.e. may either be derived from the defined template or be specified in-line. An optional type field in the matching criteria to the **receive** operation shall be used to avoid any ambiguity of the type of the value being received.

NOTE 2: Encoding attributes also participate in matching in an implicit way, by preventing the decoder to produce an abstract value from the received message encoded in a different way than specified by the attributes.

Receiving from a specific sender

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

Storing the received message and parts of the received message

If the match is successful, the value removed from the port queue and/or parts of this value can be stored in variables or formal parameters. This is denoted by the symbol '->' and the keyword **value**.

When the keyword **value** is followed by a name of a variable or formal parameter, the whole received message shall be stored in the variable or formal parameter. The variable or formal parameter shall be type compatible with the received message.

When the keyword **value** is followed by an assignment list enframed by a pair of parentheses, the whole received message and/or one or more parts of it can be stored. In a single assignment within the list, on the left hand side of the assignment symbol (":=") a field of the template type shall be referenced, on the right hand side the name of the variable or a formal parameter, in which the value shall be stored. The variable or formal parameter shall be type compatible with the type on the left hand side of the assignment symbol. As a special case the field reference can be absent to indicate that the whole message shall be stored in a variable.

Storing the sender

It is also possible to retrieve and store the component reference or address of the sender of a message. This is denoted by the keyword **sender**.

When the message is received on a connected port, only the component reference is stored in the following the **sender** keyword, but the test system shall internally store the component name too, if any (to be used in logging).

Receive any message

A **receive** operation with no argument list for the type and value matching criteria of the message to be received shall remove the message on the top of the incoming port queue (if any) if all other matching criteria are fulfilled.

Receive on any port

To **receive** a message on any port, use the **any port** keywords.

Stand-alone receive

The **receive** operation can be used as a stand-alone statement in a behaviour description. In this latter case the **receive** operation is considered to be shorthand for an **alt** statement with the **receive** operation as the only alternative.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) When defining the message in-line, the optional type part shall be present whenever the type of the message being received is ambiguous.

b) The **receive** operation shall only be used on message-based ports and the type of the value to be received shall be included in the list of incoming types of the port type definition.

c) No binding of the incoming values to the terms of the expression or to the template shall occur.

d) A message received by *receive any message* shall not be stored, i.e. the **value** clause shall not be present.

e) Type mismatch at storing the received value or parts of the received value and storing the sender shall cause an error.

f) *AddressRef* for retrieving the sending entity shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **receive** operation.

Examples

EXAMPLE 1: Basic receive

```
MyPort.receive(MyTemplate(5, MyVar));   // Matches a message that fulfils the conditions
                                        // defined by template MyTemplate at port MyPort.

MyPort.receive(A<B);    // Matches a Boolean value that depends on the outcome of A<B

MyPort.receive(integer:MyVar);  // Matches an integer value with the value of MyVar
                                // at port MyPort

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

EXAMPLE 2: Receiving from a sender, storing the message, parts of the message or the sender

```
MyPort.receive(charstring:"Hello")from MyPeer;  // Matches charstring "Hello" from MyPeer

MyPort.receive(MyType:?) -> value MyVar;    // The value of the received message is
                                            // assigned to MyVar.

MyPort.receive(MyType:?) -> value (MyVar, MyMessageIdVar:= MyType.messageId)
                            // The value of the received message is stored in the variable
                            // MyVar and the value of the messageId field of the received
                            // message is stored in the variable MyMessageIdVar.

MyPort.receive(anytype:?) -> value (MyIntegerVar := integer)
                            // If the received value is an integer, it is stored in the variable
                            // MyIntegerVar, a test case error otherwise.

MyPort.receive(charstring:?) -> value (MyCharstringVar)
                            // The received value is stored in the variable MyCharstringVar;
                            // Note that it is the same as to write "value MyCharstringVar"

MyPort.receive(A<B) -> sender MyPeer;       // The address of the sender is assigned to MyPeer

MyPort.receive(MyTemplate:{5, MyVarOne}) -> value MyVarTwo sender MyPeer;
// The received message value is stored in MyVarTwo and the sender address is stored in MyPeer.
```

EXAMPLE 3: Receive any message

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

MyPort.receive from MyPeer;             // Removes the top message from MyPort if its sender is
                                        MyPeer
```

```
MyPort.receive -> sender MySenderVar;   // Removes the top message from MyPort and assigns
                                        // the sender address to MySenderVar
```

EXAMPLE 4: Receive on any port

```
any port.receive(MyMessage);
```

22.2.3 The Trigger operation

The **trigger** operation is used to await a specific message on an incoming port queue.

Syntactical Structure

```
( Port | any port ) "." trigger
[ "(" TemplateInstance ")" ]
[ from Address ]
[ "->" [ value ( VariableRef |
                 ( "(" { VariableRef [ ":=" FieldOrTypeReference ] [","] } ")" )
               ) ]
  [ sender VariableRef ] ]
```

NOTE: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**any component**".

Semantic Description

The **trigger** operation removes the top message from the associated incoming port queue. If that top message meets the matching criteria, the **trigger** operation behaves in the same manner as a **receive** operation. If that top message does not fulfil the matching criteria, it shall be removed from the queue without any further action.

The **trigger** operation requires the port name, matching criteria for type and value, an optional **from** restriction (i.e. selection of communication partner) and an optional assignment of the matching message and sender component to variables.

Matching criteria

The matching criteria as defined in clause 22.2.2 apply also to the **trigger** operation.

Trigger on any message

A **trigger** operation with no argument list shall trigger on the receipt of any message. Thus, its meaning is identical to the meaning of receive any message.

Trigger on any port

To **trigger** on a message at any port, use the **any port** keywords.

Stand-alone trigger

The **trigger** operation can be used as a stand-alone statement in a behaviour description. In this latter case the **trigger** operation is considered to be shorthand for an **alt** statement with two alternatives (one alternative expecting the message and another alternative consuming all other messages and repeating the alt statement, see ES 201 873-4 [1]).

Storing the received message, parts of the received message or the sender

Rules in clause 22.2.2 shall apply.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The **trigger** operation shall only be used on message-based ports and the type of the value to be received shall be included in the list of incoming types of the port type definition.

b) A message received by *TriggerOnAnyMessage* shall not be assigned to a variable.

c) Type mismatch at storing the received value or parts of the received value and storing the sender shall cause an error.

d) *AddressRef* for retrieving the sending entity shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **trigger** operation.

Examples

EXAMPLE 1: Basic trigger

```
MyPort.trigger(MyType:?);
// Specifies that the operation will trigger on the reception of the first message observed of
// the type MyType with an arbitrary value at port MyPort.
```

EXAMPLE 2: Trigger from a sender and with storing message or sender

```
MyPort.trigger(MyType:?) from MyPartner;
// Triggers on the reception of the first message of type MyType at port MyPort
// received from MyPartner.

MyPort.trigger(MyType:?) from MyPartner -> value MyRecMessage;
// This example is almost identical to the previous example. In addition, the message which
// triggers i.e. all matching criteria are met, is stored in the variable MyRecMessage.

MyPort.trigger(MyType:?) -> sender MyPartner;
// This example is almost identical to the first example. In addition, the reference of the
// sender component will be retrieved and stored in variable MyPartner.

MyPort.trigger(integer:?) -> value MyVar sender MyPartner;
// Trigger on the reception of an arbitrary integer value which afterwards is stored in
// variable MyVar. The reference of the sender component will be stored in variable MyPartner.
```

EXAMPLE 3: Trigger on any message

```
MyPort.trigger;

MyPort.trigger from MyPartner;

MyPort.trigger -> sender MySenderVar;
```

EXAMPLE 4: Trigger on any port

```
any port.trigger
```

22.3 Procedure-based communication

The operations for procedure-based communication via synchronous ports are summarized in table 24.

Table 24: Overview of procedure-based communication

Communication operation	Keyword
Invoke procedure call	**call**
Accept procedure call from remote entity	**getcall**
Reply to procedure call from remote entity	**reply**
Raise exception (to an accepted call)	**raise**
Handle response from a previous call	**getreply**
Catch exception (from called entity)	**catch**
Check call/exception/reply received	**check**

22.3.1 The Call operation

The **call** operation specifies the call of a remote operation on another test component or within the SUT.

Syntactical Structure

```
Port "." call "(" TemplateInstance [ "," CallTimerValue ] ")"
[ to Address ]
```

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**all component**".

Semantic Description

The **call** operation is used to specify that a test component calls a procedure in the SUT or in another test component.

The information to be transmitted in the send part of the **call** operation is a signature that may either be defined in the form of a signature template or be defined in-line.

Handling responses and exceptions to a call

In case of non-blocking procedure-based communication the handling of exceptions to **call** operations is done by using **catch** (see clause 22.3.6) operations as alternatives in **alt** statements.

If the **nowait** option is used, the handling of responses or exceptions to **call** operations is done by using **getreply** (see clause 22.3.4) and **catch** (see clause 22.3.6) operations as alternatives in **alt** statements.

In case of blocking procedure-based communication, the handling of responses or exceptions to a call is done in the response and exception handling part of the **call** operation by means of **getreply** (see clause 22.3.4) and **catch** (see clause 22.3.6) operations.

The response and exception handling part of a **call** operation looks similar to the body of an **alt** statement. It defines a set of alternatives, describing the possible responses and exceptions to the call.

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

The response and exception handling part of a call operation is executed like an **alt** statement without any active default. This means a corresponding snapshot includes all information necessary to evaluate the (optional) Boolean guards, may include the top element (if any) of the port over which the procedure has been called and may include a timeout exception generated by the (optional) timer that supervises the call.

Handling timeout exceptions to a call

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

Nowait calls of blocking procedures

Using the keyword **nowait** instead of a timeout exception value in a **call** operation allows calling a procedure to continue without waiting either for a response or an exception raised by the called procedure or a timeout exception.

If the **nowait** keyword is used, a possible response or exception of the called procedure has to be removed from the port queue by using a **getreply** or a **catch** operation in a subsequent **alt** statement.

Calling blocking procedures without return value, out parameters, inout parameters and exceptions

A blocking procedure may have no return values, no out and inout parameters and may raise no exception. The call operation for such a procedure shall also have a response and exception handling part to handle the blocking in a uniform manner.

Calling non-blocking procedures

A non-blocking procedure has no out and inout parameters, no return value and the non-blocking property is indicated in the corresponding signature definition by means of a **noblock** keyword.

Possible exceptions raised by non-blocking procedures have to be removed from the port queue by using **catch** operations in subsequent **alt** or **interleave** statements.

Unicast, multicast and broadcast calls of procedures

Like for the **send** operation, TTCN-3 also supports unicast, multicast and broadcast calls of procedures. This can be done in the same manner as described in clause 22.2.1, i.e. the argument of the **to** clause of a **call** operation is for unicast calls the address of one receiving entity (or can be omitted in case of one-to-one connections), for multicast calls a list of addresses of a set of receivers and for broadcast calls the **all component** keyword. In case of one-to-one connections, the **to** clause may be omitted, because the receiving entity is uniquely identified by the system structure.

The handling of responses and exceptions for a blocking or non-blocking unicast **call** operation has been explained in this clause under "Handling timeout exceptions to a call". A multicast or broadcast **call** operation may cause several responses and exceptions from different communication partners.

In case of a multicast or broadcast **call** operation of a non-blocking procedure, all exceptions which may be raised from the different communication partners can be handled in subsequent **catch**, **alt** or **interleave** statements.

In case of a multicast or broadcast **call** operation of a blocking procedure, two options exist. Either, only one response or exception is handled in the response and exception handling part of the **call** operation. Then, further responses and exceptions can be handled in subsequent **alt** or **interleave** statements. Or, several responses or exceptions are handled by the use of repeat statements in one or more of the statement blocks of the response and exception handling part of the call operation: the execution of a repeat statement causes the re-evaluation of the call body.

NOTE 2: In the second case, the user needs to handle the number of repetitions.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

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

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

c) Only out parameters may be omitted or specified with a matching attribute.

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

e) A **to** clause shall be present in case of one-to-many connections.

f) *AddressRef* shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **call** operation.

g) *CallTimerValue* must be of type float.

h) The selection of the alternatives to a call shall only be based on **getreply** and **catch** operations for the called procedure. Unqualified **getreply** and **catch** operations shall only treat replies from and exceptions raised by the called procedure. The use of **else** branches and the invocation of altsteps is not allowed.

i) The evaluation of the Boolean expressions guarding the alternatives in the response and exception handling part may have side effects. In order to avoid unexpected side effects, the same rules as for the Boolean guards in **alt** statements shall be applied (see clause 20.2).

j) The call operation for a blocking procedures without return value, out parameters, inout parameters and exceptions shall also have a response and exception handling part to handle the blocking in a uniform manner.

k) In case of a multicast or broadcast **call** operation of a blocking procedure, where the **nowait** keyword is used, all responses and exceptions have to be handled in subsequent **alt** or **interleave** statements.

l) The **call** operation for a non-blocking procedure shall have no response and exception handling part, shall raise no timeout exception and shall not use the **nowait** keyword.

Examples

EXAMPLE 1: Blocking call with getreply

```
// Given …
signature MyProc (out integer MyPar1, inout boolean MyPar2);
 :
// a call of MyProc
MyPort.call(MyProc:{ -, MyVar2}) {      // in-line signature template for the call of MyProc
    [] MyPort.getreply(MyProc:{?, ?}) { }
}

// … and another call of MyProc
MyPort.call(MyProcTemplate) {           // using signature template for the call of MyProc
    [] MyPort.getreply(MyProc:{?, ?}) { }
}

MyPort.call(MyProcTemplate) to MyPeer {         // calling MyProc at MyPeer
    [] MyPort.getreply(MyProc:{?, ?}) { }
}
```

EXAMPLE 2: Blocking call with getreply and catch

```
// Given
signature MyProc3 (out integer MyPar1, inout boolean MyPar2) return MyResultType
    exception (ExceptionTypeOne, ExceptionTypeTwo);
 :

// Call of MyProc3
MyPort.call(MyProc3:{ -, true }) to MyPartner {

  [] MyPort.getreply(MyProc3:{?, ?}) -> value MyResult param (MyPar1Var,MyPar2Var) { }

  [] MyPort.catch(MyProc3, MyExceptionOne) {
        setverdict(fail);
        stop;
     }
  [] MyPort.catch(MyProc3, ExceptionTypeTwo : ?) {
        setverdict(inconc);
     }
  [MyCondition] MyPort.catch(MyProc3, MyExceptionThree) { }
}
```

EXAMPLE 3: Blocking call with timeout exception

```
MyPort.call(MyProc:{5,MyVar}, 20E-3) {

  [] MyPort.getreply(MyProc:{?, ?}) { }

  [] MyPort.catch(timeout) {                // timeout exception after 20ms
        setverdict(fail);
        stop;
     }
}
```

EXAMPLE 4: Nowait call

```
MyPort.call(MyProc:{5, MyVar}, nowait);     // The calling test component will continue
                                            // its execution without waiting for the
                                            // termination of MyProc
```

EXAMPLE 5: Blocking call without return value, out parameters, inout parameters and exceptions

```
// Given …
signature MyBlockingProc (in integer MyPar1, in boolean MyPar2);
 :
// a call of MyBlockingProc
MyPort.call(MyBlockingProc:{ 7, false }) {
  [] MyPort.getreply( MyBlockingProc:{ -, - } ) { }
}
```

EXAMPLE 6: Broadcast call

```
var boolean first:= true;
MyPort.call(MyProc:{5,MyVar}, 20E-3) to all component { // Broadcast call of MyProc
    // Handles the response from MyPeerOne
    [first] MyPort.getreply(MyProc:{?, ?}) from MyPeerOne {
        if (first) { first := false; repeat; }
        :
    }
    // Handles the response from MyPeerTwo
    [first] MyPort.getreply(MyProc:{?, ?}) from MyPeerTwo {
        if (first) { first := false; repeat; }
        :
    }
    [] MyPort.catch(timeout) {              // timeout exception after 20ms
        setverdict(fail);
        stop;
    }
}

alt {
  [] MyPort.getreply(MyProc:{?, ?}) {       // Handles all other responses to the broadcast call
        repeat
  }
}
```

EXAMPLE 7: Multicast call

```
MyPort.call(MyProc:{5,MyVar}, nowait) to (MyPeer1, MyPeer2);    // Multicast call of MyProc

interleave {
  [] MyPort.getreply(MyProc:{?, ?}) from MyPeer1 { }    // Handles the response of MyPeer1
  [] MyPort.getreply(MyProc:{?, ?}) from MyPeer2 { }    // Handles the response of MyPeer2
}
```

22.3.2 The Getcall operation

The **getcall** operation is used to accept calls.

Syntactical Structure

```
( Port | any port ) "." getcall
[ "(" TemplateInstance ")" ]
[ from Address ]
[ "->" [ param "(" { ( VariableRef ":=" ParameterIdentifier ) "," } |
                   { ( VariableRef | "-" ) "," }
                 ")" ]
       [ sender VariableRef ] ]
```

NOTE: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**any component**".

Semantic Description

The **getcall** operation is used to specify that a test component accepts a call from the SUT, or another test component.

The **getcall** operation shall remove the top call from the incoming port queue, if, and only if, the matching criteria associated to the **getcall** operation are fulfilled. These matching criteria are related to the signature of the call to be processed and the communication partner. The matching criteria for the signature may either be specified in-line or be derived from a signature template.

The assignment of **in** and **inout** parameter values to variables shall be made in the assignment part of the **getcall** operation. This allows the use of signature templates in **getcall** operations in the same manner as templates are used for types.

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

The (optional) assignment part of the **getcall** operation comprises the assignment of **in** and **inout** parameter values to variables and the retrieval of the address of the calling component. The keyword **param** is used to retrieve the parameter values of a call.

The keyword **sender** is used when it is required to retrieve the address of the sender (e.g. for addressing a **reply** or exception to the calling party in a one-to-many configuration).

Accepting any call

A **getcall** operation with no argument list for the signature matching criteria will remove the call on the top of the incoming port queue (if any) if all other matching criteria are fulfilled.

Getcall on any port

To **getcall** on any port is denoted by the **any** keyword.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The **getcall** operation shall only be used on procedure-based ports and the signature of the procedure call to be accepted shall be included in the list of allowed incoming procedures of the port type definition.

b) The signature argument of the **getcall** operation shall not be used to pass in variable names for **in** and **inout** parameters.

c) The *ParameterIdentifier*s must be from the corresponding signature definition.

d) The value assignment part shall not be used with the getcall operation.

e) Parameters of calls accepted by *accepting any call* shall not be assigned to a variable, i.e. the **param** clause shall not be present.

f) *AddressRef* for retrieving the sending entity shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **getcall** operation.

Examples

EXAMPLE 1: Basic getcall

```
MyPort.getcall(MyProc: MyProcTemplate(5, MyVar));   // accepts a call of MyProc at MyPort

MyPort.getcall(MyProc:{5, MyVar}) from MyPeer; // accepts a call of MyProc at MyPort from MyPeer
```

EXAMPLE 2: Getcall with matching and assignments of parameter values to variables

```
MyPort.getcall(MyProc:{?, ?}) from MyPartner -> param (MyPar1Var, MyPar2Var);
// The in or inout parameter values of MyProc are assigned to MyPar1Var and MyPar2Var.

MyPort.getcall(MyProc:{5, MyVar}) -> sender MySenderVar;
// Accepts a call of MyProc at MyPort with the in or inout parameters 5 and MyVar.
// The address of the calling party is retrieved and stored in MySenderVar.

// The following getcall examples show the possibilities to use matching attributes
// and omit optional parts, which may be of no importance for the test specification.

MyPort.getcall(MyProc:{5, MyVar}) -> param(MyVar1, MyVar2) sender MySenderVar;

MyPort.getcall(MyProc:{5, ?}) -> param(MyVar1, MyVar2);

MyPort.getcall(MyProc:{?, MyVar}) -> param( - , MyVar2);
// The value of the first inout parameter is not important or not used

// The following examples shall explain the possibilities to assign in and inout parameter
// values to variables. The following signature is assumed for the procedure to be called:

signature MyProc2(in integer A, integer B, integer C, out integer D, inout integer E);

MyPort.getcall(MyProc2:{?, ?, 3, - , ?}) -> param (MyVarA, MyVarB, - , -, MyVarE);
// The parameters A, B, and E are assigned to the variables MyVarA, MyVarB, and
// MyVarE. The out parameter D needs not to be considered.

MyPort.getcall(MyProc2:{?, ?, 3, -, ?}) -> param (MyVarA:= A, MyVarB:= B, MyVarE:= E);
// Alternative notation for the value assignment of in and inout parameter to variables. Note,
// the names in the assignment list refer to the names used in the signature of MyProc2
```

```
MyPort.getcall(MyProc2:{1, 2, 3, -, *}) -> param (MyVarE:= E);
// Only the inout parameter value is needed for the further test case execution
```

EXAMPLE 3: Accepting any call

```
MyPort.getcall;             // Removes the top call from MyPort.

MyPort.getcall from MyPartner;  // Removes a call from MyPartner from port MyPort

MyPort.getcall -> sender MySenderVar;   // Removes a call from MyPort and retrieves
                                        // the address of the calling entity
```

EXAMPLE 4: Getcall on any port

```
any port.getcall(MyProc:?)
```

22.3.3 The Reply operation

The **reply** operation is used to reply to a call.

Syntactical Structure

```
Port "." reply "(" TemplateInstance [ value Expression ] ")"
[ to Address ]
```

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**all component**".

Semantic Description

The **reply** operation is used to reply to a previously accepted call according to the procedure signature.

NOTE 2: The relation between an accepted call and a **reply** operation cannot always be checked statically. For testing it is allowed to specify a **reply** operation without an associated **getcall** operation.

The value part of the **reply** operation consists of a signature reference with an associated actual parameter list and (optional) return value. The signature may either be defined in the form of a signature template or it may be defined in-line.

Responses to one or more **call** operations may be sent to one, several or all peer entities connected to the addressed port. This can be specified in the same manner as described in clause 22.2.1. This means, the argument of the **to** clause of a **reply** operation is for unicast responses the address of one receiving entity, for multicast responses a list of addresses of a set of receivers and for broadcast responses the **all component** keywords.

In case of one-to-one connections, the **to** clause may be omitted, because the receiving entity is uniquely identified by the system structure.

A return value shall be explicitly stated with the **value** keyword.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) A **reply** operation shall only be used at a procedure-based port. The type definition of the port shall include the name of the procedure to which the **reply** operation belongs.

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

c) A **to** clause shall be present in case of one-to-many connections.

d) *AddressRef* shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **reply** operation.

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

Examples

```
MyPort.reply(MyProc2:{ - ,5});          // Replies to an accepted call of MyProc2.

MyPort.reply(MyProc2:{ - ,5}) to MyPeer; // Replies to an accepted call of MyProc2 from MyPeer

MyPort.reply(MyProc2:{ - ,5}) to (MyPeer1, MyPeer2); // Multicast reply to MyPeer1 and MyPeer2

MyPort.reply(MyProc2:{ - ,5}) to all component; // Broadcast reply to all entities connected
                                                // to MyPort

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

22.3.4 The Getreply operation

The **getreply** operation is used to handle replies from a previously called procedure.

Syntactical Structure

```
( Port | any port ) "." getreply
[ "(" TemplateInstance [ value TemplateInstance ]")" ]
[ from Address ]
[ "->" [ value VariableRef ]
       [ param "(" { ( VariableRef ":=" ParameterIdentifier ) "," } |
                   { ( VariableRef | "-" ) "," }
               ")" ]
       [ sender VariableRef ] ]
```

NOTE: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**any component**".

Semantic Description

The **getreply** operation is used to handle replies from a previously called procedure.

The **getreply** operation shall remove the top reply from the incoming port queue, if, and only if, the matching criteria associated to the **getreply** operation are fulfilled. These matching criteria are related to the signature of the procedure to be processed and the communication partner. The matching criteria for the signature may either be specified in-line or be derived from a signature template.

Matching against a received return value can be specified by using the **value** keyword.

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

The assignment of **out** and **inout** parameter values to variables shall be made in the assignment part of the **getreply** operation. This allows the use of signature templates in **getreply** operations in the same manner as templates are used for types.

The (optional) assignment part of the **getreply** operation comprises the assignment of **out** and **inout** parameter values to variables and the retrieval of the address of the sender of the reply. The keyword **value** is used to retrieve return values and the keyword **param** is used to retrieve the parameter values of a reply. The keyword **sender** is used when it is required to retrieve the address of the sender.

Get any reply

A **getreply** operation with no argument list for the signature matching criteria shall remove the reply message on the top of the incoming port queue (if any) if all other matching criteria are fulfilled.

If *GetAnyReply* is used in the response and exception handling part of a **call** operation, it shall only treat replies from the procedure invoked by the **call** operation.

Get a reply on any port

To get a reply on any port, use the **any port** keywords.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) A **getreply** operation shall only be used at a procedure-based port. The type definition of the port shall include the name of the procedure to which the **getreply** operation belongs.

b) The signature argument of the **getreply** operation shall not be used to pass in variable names for **out** and **inout** parameters.

c) Parameters or return values of responses accepted by *get any reply* shall not be assigned to a variable, i.e. the **param** and **value** clause shall not be present.

d) *AddressRef* for retrieving the sending entity shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **getreply** operation.

Examples

EXAMPLE 1: Basic getreply

```
MyPort.getreply(MyProc:{5, ?} value 20);    // Accepts a reply of MyProc with two out or
                                            // inout parameters and a return value of 20

MyPort.getreply(MyProc2:{ - , 5}) from MyPeer;  // Accepts a reply of MyProc2 from MyPeer
```

EXAMPLE 2: Getreply with storing inout/out parameters and return values in variables

```
MyPort.getreply(MyProc1:{?, ?} value ?) -> value MyRetValue param(MyPar1,MyPar2);
// The returned value is assigned to variable MyRetValue and the value
// of the two out or inout parameters are assigned to the variables MyPar1 and MyPar2.

MyPort.getreply(MyProc1:{?, ?} value ?) -> value MyRetValue param( - , MyPar2) sender MySender;
// The value of the first parameter is not considered for the further test execution and
// the address of the sender component is retrieved and stored in the variable MySender.

// The following examples describe some possibilities to assign out and inout parameter values
// to variables. The following signature is assumed for the procedure which has been called

signature MyProc2(in integer A, integer B, integer C, out integer D, inout integer E);

MyPort.getreply(ATemplate) -> param( - , - , - , MyVarOut1, MyVarInout1);

MyPort.getreply(ATemplate) -> param(MyVarOut1:=D, MyVarOut2:=E);

MyPort.getreply(MyProc2:{ - , - , - , 3, ?}) -> param(MyVarInout1:=E);
```

EXAMPLE 3: Get any reply

```
MyPort.getreply;                // Removes the top reply from MyPort.

MyPort.getreply from MyPeer;    // Removes the top reply received from MyPeer from MyPort.

MyPort.getreply -> sender MySenderVar;  // Removes the top reply from MyPort and retrieves the
                                        // address of the sender entity
```

EXAMPLE 4: Get a reply on any port

```
any port.getreply(Myproc:?)
```

22.3.5 The Raise operation

Exceptions are raised with the **raise** operation.

Syntactical Structure

```
Port "." raise "(" Signature "," TemplateInstance ")"
[ to Address ]
```

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**all component**".

Semantic Description

The **raise** operation is used to raise an exception.

NOTE 2: The relation between an accepted call and a **raise** operation cannot always be checked statically. For testing it is allowed to specify a **raise** operation without an associated **getcall** operation.

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

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

Exceptions to one or more **call** operations may be sent to one, several or all peer entities connected to the addressed port. This can be specified in the same manner as described in clause 22.2.1. This means, the argument of the **to** clause of a **raise** operation is for unicast exceptions the address of one receiving entity, for multicast exceptions a list of addresses of a set of receivers and for broadcast exceptions the **all component** keywords.

In case of one-to-one connections, the **to** clause may be omitted, because the receiving entity is uniquely identified by the system structure.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) An exception shall only be raised at a procedure-based port. An exception is a reaction to an accepted procedure call the result of which leads to an exceptional event.

b) The type of the exception shall be specified in the signature of the called procedure. The type definition of the port shall include in its list of accepted procedure calls the name of the procedure to which the exception belongs.

c) A **to** clause shall be present in case of one-to-many connections.

d) *AddressRef* shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **raise** operation.

Examples

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

MyPort.raise(MyProc, integer:5});   // Raises an exception with the integer value 5 for MyProc

MyPort.raise(MySignature, "My string") to MyPartner;
// Raises an exception with the value "My string" at MyPort for MySignature and
// send it to MyPartner

MyPort.raise(MySignature, "My string") to (MyPartnerOne, MyPartnerTwo);
// Raises an exception with the value "My string" at MyPort and sends it to MyPartnerOne and
// MyPartnerTwo (i.e. multicast communication)

MyPort.raise(MySignature, "My string") to all component;
// Raises an exception with the value "My string" at MyPort for MySignature and sends it
// to all entites connected to MyPort (i.e. broadcast communication)
```

22.3.6 The Catch operation

The **catch** operation is used to catch exceptions.

Syntactical Structure

```
( Port | any port ) "." catch
[ "(" ( Signature "," TemplateInstance ) | TimeoutKeyword ")" ]
[ from Address ]
[ "->" [ value ( VariableRef |
```

```
                ( "(" { VariableRef [ ":=" FieldOrTypeReference ][","] } ")" )
              ) ]
    [ sender VariableRef ] ]
```

NOTE: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**any component**".

Semantic Description

The **catch** operation is used to catch exceptions raised by a test component or the SUT as a reaction to a procedure call. Exceptions are specified as types and thus, can be treated like messages, e.g. templates can be used to distinguish between different values of the same exception type.

The **catch** operation removes the top exception from the associated incoming port queue if, and only if, that top exception satisfies all the matching criteria associated with the **catch** operation.

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

The (optional) redirection part of the **catch** operation comprises of storing the exception value and/or one or more parts of it and the retrieval of the address of the calling component. The keyword **value** is used to retrieve the value of an exception and/or the parts of it and the keyword **sender** is used when it is required to retrieve the address of the sender.

The **catch** operation may be part of the response and exception handling part of a **call** operation or be used to determine an alternative in an **alt** statement. If the **catch** operation is used in the accepting part of a **call** operation, the information about port name and signature reference to indicate the procedure that raised the exception is redundant, because this information follows from the **call** operation. However, for readability reasons (e.g. in case of complex **call** statements) this information shall be repeated.

The Timeout exception

There is one special **timeout** exception that can be caught by the **catch** operation. The **timeout** exception is an emergency exit for cases where a called procedure neither replies nor raises an exception within a predetermined time (see clause 22.3.1).

Catch any exception

A **catch** operation with no argument list allows any valid exception to be caught. The most general case is without using the **from** keyword. *CatchAnyException* will also catch the **timeout** exception.

Catch on any port

To **catch** an exception on any port use the **any** keyword.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The **catch** operation shall only be used at procedure-based ports. The type of the caught exception shall be specified in the signature of the procedure that raised the exception.

b) No binding of the incoming values to the terms of the expression or to the template shall occur. The assignment of the exception values to variables shall be made in the assignment part of the **catch** operation.

c) Catching **timeout** exceptions shall be restricted to the exception handling part of a call. No further matching criteria (including a **from** part) and no assignment part is allowed for a **catch** operation that handles a **timeout** exception.

d) Exception values accepted by *catch any exception* shall not be assigned to a variable, i.e. the **value** clause shall not be present.

e) If *CatchAnyException* is used in the response and exception handling part of a **call** operation, it shall only treat exceptions raised by the procedure invoked by the **call** operation.

f) *AddressRef* for retrieving the sending entity shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **catch** operation.

Examples

EXAMPLE 1: Basic catch

```
MyPort.catch(MyProc, integer: MyVar);   // Catches an integer exception of value
                                        // MyVar raised by MyProc at port MyPort.

MyPort.catch(MyProc, MyVar);            // Is an alternative to the previous example.

MyPort.catch(MyProc, A<B);              // Catches a boolean exception

MyPort.catch(MyProc, MyType:{5, MyVar}); // In-line template definition of an exception value.

MyPort.catch(MyProc, charstring:"Hello")from MyPeer;    // Catches "Hello" exception from MyPeer
```

EXAMPLE 2: Catch with storing value and/or sender in variables

```
MyPort.catch(MyProc, MyType:?) from MyPartner -> value MyVar;
// Catches an exception from MyPartner and assigns its value to MyVar.

MyPort.catch(MyProc, MyTemplate(5)) -> value MyVarTwo sender MyPeer;
// Catches an exception, assigns its value to MyVarTwo and retrieves the
// address of the sender.

MyPort.catch(MyProc, MyTemplate(5)) -> value (MyVarThree:= f1)
                                       sender MyPeer;
// Catches an exception, assigns the value of its field f1 to MyVarThree and retrieves the
// address of the sender.
```

EXAMPLE 3: The Timeout exception

```
MyPort.call(MyProc:{5,MyVar}, 20E-3) {
  [] MyPort.getreply(MyProc:{?, ?}) { }
  [] MyPort.catch(timeout) {                // timeout exception after 20ms
        setverdict(fail);
        stop;
     }
}
```

EXAMPLE 4: Catch any exception

```
MyPort.catch;

MyPort.catch from MyPartner;

MyPort.catch -> sender MySenderVar;
```

EXAMPLE 5: Catch on any port

```
any port.catch;
```

22.4 The Check operation

The **check** operation allows reading the top element of a message-based or procedure-based *incoming* port queue.

Syntactical Structure

```
( Port | any port ) "." check
[ "("
     ( PortReceiveOp | PortGetCallOp | PortGetReplyOp | PortCatchOp ) |
     ( [ from Address ] [ "->" sender VariableRef ] )
")" ]
```

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**any component**".

Semantic Description

The **check** operation is a generic operation that allows read access to the top element of message-based and procedure-based *incoming* port queues without removing the top element from the queue. The **check** operation has to handle values of a certain type at message-based ports and to distinguish between calls to be accepted, exceptions to be caught and replies from previous calls at procedure-based ports.

The receiving operations **receive**, **getcall**, **getreply** and **catch** together with their matching and value, sender or parameter storing parts, are used by the **check** operation to define the conditions that have to be checked and the information to be optionally extracted.

It is the *top* element of an incoming port queue that shall be checked (it is not possible to look *into* the queue). If the queue is empty the **check** operation fails. If the queue is not empty, a copy of the top element is taken and the receiving operation specified in the **check** operation is performed on the copy. The **check** operation fails if the receiving operation fails i.e. the matching criteria are not fulfilled. In this case the *copy* of the top element of the queue is discarded and test execution continues in the normal manner, i.e. the statement or alternative next to the check operation is evaluated. The **check** operation is successful if the receiving operation is successful. In this case, the value, sender or parameter storing parts of the receiving operation, if any, are executed, i.e. the message and/or a part of it, the sender's address or component reference, the parameter(s) of the call or reply or the value of the exception are stored in the associated variables.

If **check** is used as a stand-alone statement, it is considered to be a shorthand for an **alt** statement with the **check** operation as the only alternative.

Check any operation

A **check** operation with no argument list allows checking whether something waits for processing in an incoming port queue. The **check** any operation allows to distinguish between different senders (in case of one-to-many connections) by using a **from** clause and to retrieve the sender by using a shorthand assignment part with a **sender** clause.

NOTE 2: Information related to the message-based input queue of a mixed port can be retrieved easily by using the **check** operation in combination with a **receive** any operation, e.g. `MyPort.check(receive) -> sender Mysender`.

Check on any port

To **check** on any port, use the **any port** keywords.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

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

b) *AddressRef* for retrieving the sending entity shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **check** operation.

NOTE 3: In most cases the correct usage of the check operation can be checked statically, i.e. before/during compilation.

Examples

EXAMPLE 1: Basic check

```
MyPort1.check(receive(5));  // Checks for an integer message of value 5.

MyPort1.check(receive(charstring:?) -> value MyCharVar);
// Checks for a charstring message and stores the message if the message type is charstring

MyPort2.check(getcall(MyProc:{5, MyVar}) from MyPartner);
// Checks for a call of MyProc at port MyPort2 from MyPartner

MyPort2.check(getreply(MyProc:{5, MyVar} value 20));
// Checks for a reply from procedure MyProc at MyPort2 where the returned value is 20 and
// the values of the two out or inout parameters are 5 and the value of MyVar.
```

```
MyPort2.check(catch(MyProc, MyTemplate(5, MyVar)));
MyPort2.check(getreply(MyProc1:{?, MyVar} value *) -> value MyReturnValue param(MyPar1,-));
MyPort.check(getcall(MyProc:{5, MyVar}) from MyPartner -> param (MyPar1Var, MyPar2Var));
MyPort.check(getcall(MyProc:{5, MyVar}) -> sender MySenderVar);
```

EXAMPLE 2: Check any operation

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

EXAMPLE 3: Check on any port

```
any port.check;
```

22.5 Controlling communication ports

TTCN-3 operations for controlling message-based and procedure-based ports are presented in table 25.

Table 25: Overview of TTCN-3 port operations

Port operations	
Statement	**Associated keyword or symbol**
Clear port	**clear**
Start port	**start**
Stop port	**stop**
Halt port	**halt**

22.5.1 The Clear port operation

The **clear** port operation empties incoming port queues.

Syntactical Structure

```
( Port | ( all port ) ) "." clear
```

Semantic Description

The **clear** operation removes the contents of the *incoming* queue of the specified port or of all ports of the test component performing the **clear** operation.

If a port queue is already empty then this operation shall have no action on that port.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

```
MyPort.clear;   // clears port MyPort
```

22.5.2 The Start port operation

The **start** operation enables sending and receiving operations on the port(s).

Syntactical Structure

```
( Port | ( all port ) ) "." start
```

Semantic Description

If a port is defined as allowing receiving operations such as **receive**, **getcall** etc., the **start** operation clears the incoming queue of the named port and starts listening for traffic over the port. If the port is defined to allow sending operations then the operations such as **send**, **call**, **raise** etc., are also allowed to be performed at that port.

By default, all ports of a component shall be started implicitly when a component is created. The start port operation will cause unstopped ports to be restarted by removing all messages waiting in the incoming queue.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

```
MyPort.start;   // starts MyPort
```

22.5.3 The Stop port operation

The **stop** operation disables sending and disallow receiving operations to match at the port(s).

Syntactical Structure

```
( Port | ( all port ) ) "." stop
```

Semantic Description

If a port is defined as allowing receiving operations such as **receive** and **getcall,** the **stop** operation causes listening at the named port to cease. If the port is defined to allow sending operations then **stop** port disallows the operations such as **send**, **call**, **raise** etc., to be performed.

To cease listening at the port means that all receiving operations defined before the stop operation shall be completely performed before the working of the port is suspended.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

```
MyPort.receive (MyTemplate1) -> value RecPDU;
                                    // the received value is decoded, matched against
                                    // MyTemplate1 and the matching value is stored
                                    // in the variable RecPDU
MyPort.stop;                        // No receiving operation defined following the stop
                                    // operation is executed (unless the port is restarted
                                    // by a subsequent start operation)
MyPort.receive (MyTemplate2);       // This operation does not match and will block (assuming
                                    // that no default is activated)
```

22.5.4 The Halt port operation

The **halt** operation is comparable to the **stop** operation, but allows entries being already in the queue to be processed with receiving operations.

Syntactical Structure

```
( Port | ( all port ) ) "." halt
```

Semantic Description

If a port allows receiving operations such as **receive**, **trigger** and **getcall,** the **halt** operation disallows receiving operations to succeed for messages and procedure call elements that enter the port queue after performing the **halt** operation at that port. Messages and procedure call elements that were already in the queue before the **halt** operation can still be processed with receiving operations. If the port allows sending operations then **halt** port immediately disallows sending operations such as **send**, **call**, **raise** etc. to be performed. Subsequent halt operations have no effect on the state of the port or its queue.

NOTE 1: The port **halt** operation virtually puts a marker after the last entry in the queue received when the operation is performed. Entries ahead of the marker can be processed normally. After all entries in the queue ahead of the marker have been processed, the state of the port is equivalent to the stopped state.

NOTE 2: If a port **stop** operation is performed on a halted port before all entries in the queue ahead of the marker have been processed, further receive operations are disallowed immediately (i.e. the marker is virtually moved to the top of the queue).

NOTE 3: A port **start** operation on a halted port clears all entries in the queue irrespectively if they arrived before or after performing the port **halt** operation. It also removes the marker.

NOTE 4: A port **clear** operation on a halted port clears all entries in the queue irrespectively if they arrived before or after performing the port **halt** operation. It also virtually puts the marker at the top of the queue.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

```
MyPort.halt;                          // No sending allowed on Myport from this moment on;
                                      // processing of messages in the queue still possible.
MyPort.receive (MyTemplate1);         // If a message was already in the queue before the halt
                                      // operation and it matches MyTemplate1, it is processed;
                                      // otherwise the receive operation blocks.
```

22.6 Use of any and all with ports

The keywords **any** and **all** may be used with configuration and communication operations as indicated in table 26.

Table 26: Any and All with ports

Operation	Allowed		Example
	any	all	
receive, trigger, getcall, getreply, catch, check)	yes		**any port.receive**
connect / map			
disconnect / unmap		yes	**unmap**(**self : all port**)
start, stop, clear, halt		yes	**all port.start**

NOTE: Ports are owned by test components and instantiated when a component is created. The keywords **any port** and **all port** address all ports owned by a test component and not only the ports known in the scope of the function or altstep that is executed on the component.

23 Timer operations

TTCN-3 supports a number of timer operations as given in table 27. These operations may be used in test cases, functions, altsteps and module control.

Table 27: Overview of TTCN-3 timer operations

Timer operations	
Statement	**Associated keyword or symbol**
Start timer	**start**
Stop timer	**stop**
Read elapsed time	**read**
Check if timer running	**running**
Timeout event	**timeout**

23.1 The timer mechanism

It is assumed that each test component and the module control maintain their own *running-timers list* and *timeout-list*, i.e. a list of all timers that are actually running and a list of all timers that have timed out. The timeout-lists are part of the snapshots that are taken when a test case is executed. The running-timers list and timeout-list of a component or module control are updated if a timer of the component or module control is started, is stopped, times out or the component or module control executes a **`timeout`** operation.

NOTE 1: The running-timers list and the timeout-list are only a conceptual lists and do not restrict the implementation of timers. Other data structures like a set, where the access to timeout events is not restricted by, e.g. the order in which the timeout events have happened, may also be used.

NOTE 2: Conceptually, each test component and module control maintain one running-timers list and one timeout-list only. However, within a given scope unit only timers known in the scope unit can be accessed individually, i.e. timers that are declared in the scope unit, passed in as parameters to the scope unit or known via a runs-on clause. In some special cases (e.g. for re-establishing a test component during a test run), it can be necessary to stop timers local to other scope units or to check if timers local to other scope units are running or have already timed out. This can be done by using the keywords **`all`** and **`any`** in combination with the timer operations **`stop`**, **`timeout`** and **`running`**. Allowed combinations are defined in clause 23.7.

When a timer expires, the timer becomes immediately inactive. A timeout event is placed in the timeout-list and the timer is removed from the running-timer list of the test component or module control for which the timer has been declared. Only one entry for any particular timer may appear in the timeout-list and running-timer list of the test component or module control for which the timer has been declared.

All running timers shall automatically be cancelled when a test component is explicitly or implicitly stopped.

23.2 The Start timer operation

The **`start`** timer operation is used to indicate that a timer shall start running.

Syntactical Structure

```
( ( TimerIdentifier | TimerParIdentifier ) { "[" SingleExpression "]" } )
"." start [ "(" TimerValue ")" ]
```

Semantic Description

When a timer is started, its name is added to the list of running timers (for the given scope unit).

The optional timer value parameter shall be used if no default duration is given, or if it is desired to override the default value specified in the timer declaration. When a timer duration is overridden, the new value applies only to the current instance of the timer, any later **`start`** operations for this timer, which do not specify a duration, shall use the default duration.

Starting a timer with the timer value 0.0 means that the timer times out immediately. Starting a timer with a negative timer value, e.g. the timer value is the result of an expression, or without a specified timer value shall cause a runtime error.

The timer clock runs from the float value zero (0.0) up to maximum stated by the duration parameter.

The **start** operation may be applied to a running timer, in which case the timer is stopped and re-started. Any entry in a timeout-list for this timer shall be removed from the timeout-list.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) Timer value shall be a non-negative numerical **float** number (i.e. the value shall be greater or equal 0.0, infinity and not_a_number are disallowed).

Examples

```
MyTimer1.start;          // MyTimer1 is started with the default duration
MyTimer2.start(20E-3);   // MyTimer2 is started with a duration of 20 ms

// Elements of timer arrays may also be started in a loop, for example
timer t_Mytimer [5];
var float v_timerValues [5];

for (var integer i := 0; i<=4; i:=i+1)
  { v_timerValues [i] := 1.0 }

for (var integer i := 0; i<=4; i:=i+1)
  {t_Mytimer [i].start ( v_timerValues [i])}
```

23.3 The Stop timer operation

The **stop** operation is used to stop a running timer.

Syntactical Structure

```
( ( ( TimerIdentifier | TimerParIdentifier ) { "[" SingleExpression "]" } ) |
      all timer )
"." stop
```

Semantic Description

A **stop** operation removes a running timer from the list of running timers. A stopped timer becomes inactive and its elapsed time is set to the float value zero (0.0).

Stopping an inactive timer is a valid operation, although it does not have any effect. Stopping an expired timer causes the entry for this timer in the timeout-list to be removed.

The **all** keyword may be used to stop all timers that have been started on a component or module control.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

```
MyTimer1.stop;      // stops MyTimer1
all timer.stop;     // stops all running timers
```

23.4 The Read timer operation

The **read** operation is used to retrieve the time that has elapsed since the specified timer was started.

Syntactical Structure

```
( ( TimerIdentifier | TimerParIdentifier ) { "[" SingleExpression "]" } )
"." read
```

Semantic Description

The **read** operation returns the time that has elapsed since the specified timer was started. The returned value shall be of type **float**.

Applying the **read** operation on an inactive timer, i.e. on a timer not listed on the running-timer list, will return the float value zero (0.0).

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

```
var float Myvar;
MyVar := MyTimer1.read; // assign to MyVar the time that has elapsed since MyTimer1 was started
```

23.5 The Running timer operation

The **running** timer operation is used to check whether a timer is in the running-timer list.

Syntactical Structure

```
( ( ( TimerIdentifier | TimerParIdentifier ) { "[" SingleExpression "]" } ) |
      any timer )
"." running
```

Semantic Description

The **running** timer operation is used to check whether a specific timer visible in the given scope unit is listed on the running-timer list or not (i.e. that it has been started and has neither timed out nor been stopped). The operation returns the value **true** if the timer is listed on the list, **false** otherwise.

The **any** keyword may be used to check if any timer started on a component or module control is running.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

EXAMPLE 1: Checking if a specific timer is running

```
if (MyTimer1.running) { … }
```

EXAMPLE 2: Checking if an arbitrary timer is running

```
if (any timer.running) { … }
```

23.6 The Timeout operation

The **timeout** operation allows to check the expiration of timers.

Syntactical Structure

```
( ( ( TimerIdentifier | TimerParIdentifier ) { "[" SingleExpression "]" } ) |
      any timer )
```

```
"." timeout
```

Semantic Description

The **timeout** operation allows to check the expiration of a specific timer in the scope unit of a test component or module control in which the timeout operation has been called or of any timer that has been started on a test component or module control before entering the scope in which the **timeout** operation has been called.

When a **timeout** operation is processed, if a timer name is indicated, the timeout-list is searched according to the TTCN-3 scope rules. If there is a timeout event matching the timer name, that event is removed from the timeout-list, and the **timeout** operation succeeds.

The **timeout** can be used to determine an alternative in an **alt** statement or as stand-alone statement in a behaviour description. In the latter case a **timeout** operation is considered to be shorthand for an **alt** statement with the **timeout** operation as the only alternative.

The **any** keyword used with the **timeout** operation succeeds if the timeout-list is not empty.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The **timeout** shall not be used in a **boolean** expression.

Examples

EXAMPLE 1: Timeout of a specific timer

```
MyTimer1.timeout;   // checks for the timeout of the previously started timer MyTimer1
```

EXAMPLE 2: Timeout of an arbitrary timer

```
any timer.timeout; // checks for the timeout of any previously started timer
```

23.7 Summary of use of any and all with timers

The keywords **any** and **all** may be used with timer operations as indicated in table 28.

Table 28: Any and All with Timers

Operation	Allowed		Example
	any	all	
start			
stop		yes	**all timer.stop**
read			
running	yes		**if (any timer.running) {…}**
timeout	yes		**any timer.timeout**

24 Test verdict operations

Verdict operations given in table 29 allow to set and retrieve verdicts. These operations shall only be used in test cases, altsteps and functions.

Table 29: Overview of TTCN-3 test verdict operations

Test verdict operations	
Statement	**Associated keyword or symbol**
Set local verdict	**setverdict**
Get local verdict	**getverdict**

24.1 The Verdict mechanism

Each test component of the active configuration shall maintain its own local verdict. The local verdict is an object which is created for each test component at the time of its creation. It is used to track the individual verdict in each test component (i.e. in the MTC and in each and every PTC).

Additionally, there is a global test case verdict instantiated and handled by the test system that is updated when each test component (i.e. the MTC and each and every PTC) terminates execution (see figure 14). This verdict is not accessible to the **getverdict** and **setverdict** operations. The value of this verdict shall be returned by the test case when it terminates execution. If the returned verdict is not explicitly saved in the control part (e.g. assigned to a variable) then it is lost.

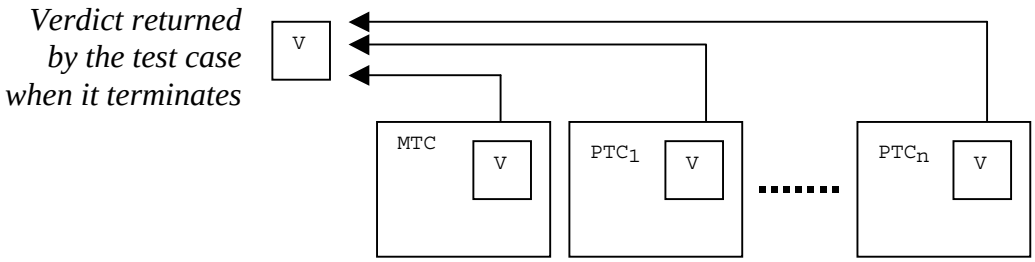


Figure 14: Illustration of the relationship between verdicts

NOTE 1: TTCN-3 does not specify the actual mechanisms that perform the updating of the local and test case verdicts. These mechanisms are implementation dependent.

The verdict can have five different values: **pass**, **fail**, **inconc**, **none** and **error**, i.e. the distinguished values of the **verdicttype** (see clause 6.1).

NOTE 2: **inconc** means an inconclusive verdict.

When a test component is instantiated, its local verdict object is created and set to the value **none**.

When changing the value of the local verdict (i.e. using the **setverdict** operation) the effect of this change shall follow the overwriting rules listed in table 30. The test case verdict is implicitly updated on the termination of a test component. The effect of this implicit operation shall also follow the overwriting rules listed in table 30.

Table 30: Overwriting rules for the verdict

Current value of Verdict	New verdict assignment value			
	pass	inconc	fail	none
None	pass	inconc	fail	none
Pass	pass	inconc	fail	pass
Inconc	inconc	inconc	fail	inconc
Fail	fail	fail	fail	fail

The **error** verdict is special in that it is set by the test system to indicate that a test case (i.e. run-time) error has occurred. It shall not be set by the **setverdict** operation and will not be returned by the getverdict operation. No other verdict value can override an **error** verdict. This means that an **error** verdict can only be a result of an **execute** test case operation.

Together with the local test verdict, each test component shall also maintain an implicit **charstring** variable to store information about the reasons for assigning the verdict. The implicit **charstring** variable shall have no effect on the overwriting rules and on the calculation of the final test case verdict. On the termination of the test component, the local verdict of the test component shall be logged together with the implicit **charstring** variable. The implicit **charstring** variable cannot be retrieved and read by any TTCN-3 function, it only provides additional information for logging.

24.2 The Setverdict operation

The local verdict is set with the **setverdict** operation.

Syntactical Structure

```
setverdict "(" SingleExpression { "," ( FreeText | TemplateInstance ) } ")"
```

Semantic Description

The value of the local verdict is changed with the **setverdict** operation. The effect of this change shall follow the overwriting rules listed in table 30.

The optional parameters allow to provide information that explain the reasons for assigning the verdict. This information is composed to a string and stored in an implicit **charstring** variable. On termination of the test component, the actual local verdict is logged together with the implicit **charstring** variable. Since the optional parameters can be seen as log information, the same rules and restrictions as for the parameters of the log statement (clause 19.11) apply.

As the result of the setverdict operation, the implicit **charstring** variable is overwritten whenever the local verdict of a test component is overwritten. A **setverdict** operation with a verdict only that overwrites the current local verdict, will also clear the implicit **charstring** variable. This means previously stored information gets lost.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The **setverdict** operation shall only be used with the values **pass**, **fail**, **inconc** and **none**. It shall not be used to assign the value **error**, this is set by the test system only to indicate run-time errors.

b) *SingleExpression* shall resolve to a value of type **verdict**.

c) For *FreeText* and *TemplateInstance*, the same rules and restrictions apply as for the parameters of the **log** statement. Table 18 lists all language elements that can be used in a setverdict operation.

Examples

EXAMPLE 1:

```
setverdict(pass);   // the local verdict is set to pass
:
setverdict(fail);   // until this line is executed, which will result in the value
                    // of the local verdict being overwritten to fail
                    // When the ptc terminates the test case verdict is set to fail
```

EXAMPLE 2:

```
var integer myVar:= 1;
:
MyPort.receive(integer:MyVar);  // Matches an integer value with the value of MyVar
                                // at port MyPort
setverdict(pass, "Value received: ", myVar ); // Provided the actual test component verdict is
                                // none: local verdict is set to pass, the implicit
                                // charstring variable is set to "Value received: 5"
stop;                           // The test component terminates. The local test verdict and
                                // implicit charstring variable are logged
```

24.3 The Getverdict operation

The value of the local verdict may be retrieved using the **getverdict** operation.

Syntactical Structure

```
getverdict
```

Semantic Description

The **getverdict** operation returns the actual value of the local verdict.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

```
MyResult := getverdict; // Where MyResult is a variable of type verdicttype
```

25 External actions

In some testing situations some interface(s) to the SUT may be missing or unknown a priori (e.g. management interface) but it may be necessary that the SUT is stimulated to carry out certain actions (e.g. send a message to the test system). Also certain actions may be required from the test executing personnel (e.g. to change the environmental conditions of testing like the temperature, voltage of the power feeding, etc.).

The required action may be described as a string expression, i.e. the use of literal strings, string typed variables and parameters, etc. and any concatenation thereof are allowed.

Syntactical Structure

```
action "(" { ( FreeText | Expression ) ["&"] } ")"
```

Semantic Description

External actions can be used in test cases, functions, altsteps and module control.

There is no specification of what is done to or by the SUT to trigger this action, only an informal description of the required action itself.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) *Expression* shall have the base type charstring or universal charstring.

Examples

```
var charstring myString:= " now."
action("Send MyTemplate on lower PCO" & myString);  // Informal description of the
                                                    // external action
```

26 Module control

Test cases are defined in the module definitions part while the module control part manages their execution. The statements and operations that can be used in the module control are summarized in table 31.

Table 31: Overview of TTCN-3 statements and operations in module control

Statement	Associated keyword or symbol
Assignments	**:=**
If-else	**if (…) {…} else {…}**
Select case	**select case (…) { case (…) {…} case else {…}}**
For loop	**for (…) {…}**
While loop	**while (…) {…}**
Do while loop	**do {…} while (…)**
Label and Goto	**label / goto**
Stop execution	**stop**
Leaving a loop, alt or interleave	**break**
Next iteration of a loop	**continue**
Logging	**log**

Statement	Associated keyword or symbol
Alternative behaviour (see note)	**alt {…}**
Re-evaluation of alternative behaviour (see note)	**repeat**
Interleaved behaviour (see note)	**interleave {…}**
Activate a default (see note)	**activate**
Deactivate a default (see note)	**deactivate**
Start timer	**start**
Stop timer	**stop**
Read elapsed time	**read**
Check if timer running	**running**
Timeout event	**timeout**
Stimulate an (SUT) action externally	**action**
Execute test case	**execute**
NOTE: Can be used to control timer operations only.	

26.1 The Execute statement

Test cases are executed with an **`execute`** statement in the module control.

Syntactical Structure

```
execute "(" TestcaseRef "(" [ { ActualPar  ["," ] } ] ")" [ "," TimerValue [ "," HostId ] ] ")"
```

Semantic Description

In the module control part the **`execute`** statement is used to start test cases (see clause 27.1). The result of an executed test case is always a value of type **`verdicttype`**. Every test case shall contain one and only one MTC the type of which is referenced in the header of the test case definition. The behaviour defined in the test case body is the behaviour of the MTC.

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

Test case start

A test case is called using an **`execute`** statement. As the result of the execution of a test case, a test case verdict of either **`none`**, **`pass`**, **`inconc`**, **`fail`** or **`error`** shall be returned and may be assigned to a variable for further processing.

Optionally, the **`execute`** statement allows supervision of a test case by means of a timer duration.

Also optionally, the execute statement allows deployment of the MTC to a specific host before starting the execution. The host is identified by means of a host id.

Test case parameterization and configuration

All variables (if any) defined in the control part of a module shall be passed into the test case by parameterization if they are to be used in the behaviour definition of that test case, i.e. TTCN-3 does not support global variables of any kind.

At the start of each test case, the test configuration shall be reset. This means that all components and ports conducted by **`create`**, **`connect`**, etc. operations in a previous test case were destroyed when that test case was stopped (hence are not "visible" to the new test case).

Test case termination

A test case terminates with the termination of the MTC. On termination of the MTC (explicitly or implicitly), all running parallel test components shall be removed by the test system.

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

The final verdict of a test case is calculated based on the final local verdicts of the different test components according to the rules defined in clause 24.1. The actual local verdict of a test component becomes its final local verdict when the test component terminates itself or is stopped by itself, another test component or by the test system.

NOTE 2: To avoid race conditions for the calculation of test verdicts due to the delayed stopping of PTCs, the MTC should ensure that all PTCs have stopped (by means of the **done** or **killed** statement) before it stops itself.

Test case timer

Timer may be used to supervise the execution of a test case. This may be done using an explicit timeout in the **execute** statement. If the test case does not end within this duration, the result of the test case execution shall be an error verdict and the test system shall terminate the test case. The timer used for test case supervision is a system timer and need not be declared or started.

Host id

A host id can be used to give a specific deployment location to the test system where the MTC shall be started and execute its behaviour. If a host id is provided, the execute statement shall end with a test case error if the MTC cannot be deployed on the specified host.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) The *TimerValue* shall resolve to a non-negative numerical float value (i.e. the value shall be greater or equal 0.0, infinity and not_a_number are disallowed).

b) When the corresponding formal parameter is not of template type *TemplateInstance* shall resolve to an *Expression*.

c) The execute statement shall not be called from within an existing executing testcase or function chain called from a test case, i.e. test cases can only be executed from the control part or from functions directly called from the control part.

d) The *HostId* parameter shall resolve to a **charstring** value.

Examples

EXAMPLE 1: Test case execution without keeping the test case verdict

```
execute(MyTestCase1());                    // executes MyTestCase1, without storing the
                                           // returned test verdict and without time
                                           // supervision
```

EXAMPLE 2: Test case execution with keeping the test case verdict

```
MyVerdict := execute(MyTestCase2());       // executes MyTestCase2 and stores the resulting
                                           // verdict in variable MyVerdict
```

EXAMPLE 3: Test case timer

```
MyVerdict := execute(MyTestCase3(),5E-3);  // executes MyTestCase3 and stores the resulting
                                           // verdict in variable MyVerdict. If the test case
                                           // does not terminate within 5ms, MyVerdict will
                                           // get the value 'error'

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

EXAMPLE 4: Host id

```
MyVerdict := execute(MyTestCase3(), -, "Host1");
                                           // executes MyTestCase3 with unlimited time
                                           // with MTC deployed to 'Host1'
```

26.2 The Control part

The control part defines, in which order, sequence, loop, under which preconditions, and with which parameters test cases are to be executed.

Syntactical Structure

```
control "{"
     { ( ConstDef |
         TemplateDef |
         VarInstance |
         TimerInstance |
         TimerStatements |
         BasicStatements |
         BehaviourStatements |
         SUTStatements |
         stop ) [";"] }
"}"
[ WithStatement ] [";"]
```

Semantic Description

Sequence of test cases

Program statements specify such things like the order in which test cases are to be executed or the number of times a test case should run.

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

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

Timer operations may also be used explicitly to control test case execution.

Selection/deselection of test cases

The selection and deselection of test cases can also be used to control the execution of test cases.

There are different ways in TTCN-3 to select and deselect test cases. For example, boolean expressions may be used to select and deselect which test cases are to be executed. This includes, of course, the use of functions that return a **`boolean`** value.

Another way to execute test cases as a group is to collect them in a function and execute that function from the module control.

As a test case returns a single value of type **`verdicttype`**, it is also possible to control the order of test case execution depending on the outcome of a test case. The use of the TTCN-3 verdicttype is another way to select test cases.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) Configuration statements such as connect and map (with the exception of stop execution, which is allowed), communication statements such as send and receive and verdict statements such as setverdict shall not be used in the control part.

b) Statements for alternative behaviours shall only be used to control timer behaviours.

c) The restrictions on the use of statements in the control part are given in table 16.

Examples

EXAMPLE 1: Test case execution in a loop

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

EXAMPLE 2: Test case execution controlled by a timer and a counter

```
// Example of the use of the running timer operation
while (T1.running or x<10)  // Where T1 is a previously started timer
{   execute(MyTestCase());
    x := x+1;
}

// Example of the use of the start and timeout operations

timer T1 := 1.0;
 :
execute(MyTestCase1());
T1.start;
T1.timeout; // Pause before executing the next test case
execute(MyTestCase2());
```

EXAMPLE 3: Selection/deselection of test cases with Boolean expressions

```
module MyTestSuite () {
     :
    control {
         :
        if (MySelectionExpression1()) {
            execute(MySimpleTestCase1());
            execute(MySimpleTestCase2());
            execute(MySimpleTestCase3());
        }
        if (MySelectionExpression2()) {
            execute(MySimpleTestCase4());
            execute(MySimpleTestCase5());
            execute(MySimpleTestCase6());
        }
         :
    }
}
```

EXAMPLE 4: Selection/deselection of test cases with functions

```
function MyTestCaseGroup1()
{   execute(MySimpleTestCase1());
    execute(MySimpleTestCase2());
    execute(MySimpleTestCase3());
}
function MyTestCaseGroup2()
{   execute(MySimpleTestCase4());
    execute(MySimpleTestCase5());
    execute(MySimpleTestCase6());
}
 :
control
{   if (MySelectionExpression1()) { MyTestCaseGroup1(); }
    if (MySelectionExpression2()) { MyTestCaseGroup2(); }
     :
}
```

EXAMPLE 5: Selection/deselection of test cases based on test case verdicts

```
if ( execute (MySimpleTestCase()) == pass )
   { execute (MyGoOnTestCase()) }
```

```
else
   { execute (MyErrorRecoveryTestCase()) };
```

27 Specifying attributes

TTCN-3 uses attributes to give special characterization/meaning to language elements such as specific presentation format, specific encoding and encoding variants, and user-defined properties.

27.1 The Attribute mechanism

Attributes can be associated with TTCN-3 language elements by means of the with statement.

27.1.1 Scope of attributes

A **with** statement may associate attributes to a single language element or to elements or fields of structured types (in a recursive way), the same way as specified in clauses 6.2.1.1 and 6.2.3.2. It is also possible to associate attributes to a number of language elements by, e.g. listing fields of a structured type in an attribute statement associated with a single type definition or associating a **with** statement to the surrounding scope unit or **group** of language elements.

EXAMPLE 1: // attributes for single language elements and groups

```
// MyPDU1 will be displayed as PDU
type record MyPDU1 { … } with { display "PDU"}

// MyPDU2 will be displayed as PDU with the application specific extension attribute MyRule
type record MyPDU2 { … }
with
{
    display "PDU";
    extension "MyRule"
}

// The following group definition …
group MyPDUs {
    type record MyPDU3 { … }
    type record MyPDU4 { … }
}
with {display "PDU"}    // All types of group MyPDUs will be displayed as PDU

// is identical to
group MyPDUs {
    type record MyPDU3 { … } with { display "PDU"}
    type record MyPDU4 { … } with { display "PDU"}
}
```

EXAMPLE 2: // attributes for fields and elements

```
type record MyRec {
    integer field1,
    record {
        integer eField1,
        boolean eField2
    } field2
}
with { display (field2.eField1) "colour blue" }
// the embedded field eField1 is displayed blue

type record of integer MyRecOfInteger
with { display ([-]) "colour green"
// all integer elements are displayed green

type record of integer MyRecOfInteger2
with { display ([-]) "colour red" }
// integer elements are displayed red

const MyRecOfInteger c_MyRecordOfInt := {0, 1, 2, 3}
with { display ([0]) "colour blue" }
// the first element is displayed blue, the other elements are displayed red
```

27.1.2 Overwriting rules for attributes

An attribute definition in a lower scope unit will override a general attribute definition in a higher scope. Additional overwriting rules for variant attributes are defined in the present clause.

EXAMPLE 1:

```
type record MyRecordA
{
    :
} with { encode "RuleA" }

// In the following, MyRecordA is encoded according to RuleA and not according to RuleB
type record MyRecordB
{
    :
    field   MyRecordA
} with { encode "RuleB" }
```

A **with** statement that is placed inside the scope of another **with** statement shall override the outermost **with**. This shall also apply to the use of the **with** statement with groups. Care should be taken when the overwriting scheme is used in combination with references to single definitions. The general rule is that attributes shall be assigned and overwritten according to the order of their occurrence.

```
// Example of the use of the overwriting scheme of the with statement
group MyPDUs
{
    type record MyPDU1 { … }
    type record MyPDU2 { … }

    group MySpecialPDUs
    {
        type record MyPDU3 { … }
        type record MyPDU4 { … }
    }
    with {extension "MySpecialRule"}    // MyPDU3 and MyPDU4 will have the application
                                        // specific extension attribute MySpecialRule
}
with
{
    display "PDU";      // All types of group MyPDUs will be displayed as PDU and
    extension "MyRule"; // (if not overwritten) have the extension attribute MyRule
}

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

An attribute definition in a lower scope can be overwritten in a higher scope by using the **override** directive.

EXAMPLE 2:

```
type record MyRecordA
{
    :
} with { encode "RuleA" }

// In the following, MyRecordA is encoded according to RuleB
type record MyRecordB
{
    :
    fieldA  MyRecordA
} with { encode override "RuleB" }
```

The **override** directive forces all contained types at all lower scopes to be forced to the specified attribute.

An attribute definition for a field or element of a structured type overrides the corresponding attribute of the structured type, as regards the identified field or element. The attribute definition for a field or element of a structured type can however be overridden with the override directive in the attribute definition of the structured type.

27.1.2.1 Additional overwriting rules for variant attributes

A **variant** attribute is always related to an **encode** attribute. Whereas a variant of an encoding may change, an encoding shall not change without overwriting all current variant attributes. Therefore, for variant attributes the following overwriting rules apply:

- a **variant** attribute overwrites an current **variant** attribute according to the rules defined in clause 27.1.2;
- an **encoding** attribute, which overwrites a current **encoding** attribute according to the rules defined in clause 27.1.2, also overwrites a corresponding current **variant** attribute, i.e. no new **variant** attribute is provided, but the current **variant** attribute becomes inactive;
- an **encoding** attribute, which changes a current **encoding** attribute of an imported language element according to the rules defined in clause 27.1.3, also changes a corresponding current **variant** attribute, i.e. no new **variant** attribute is provided, but the current **variant** attribute becomes inactive.

EXAMPLE:

```
module MyVariantEncodingModule {
    :
    type charstring MyCharString;   // Normally encoded according to "Encoding 1"
    :
    group MyVariantsOne {
        :
        type record MyPDUone
        {
            integer     field1, // field1 will be encoded according to "Encoding 2" only.
                                // "Encoding 2" overwrites "Encoding 1" and variant "Variant 1"
            Mytype      field3  // field3 will be encoded according to "Encoding 1" with
                                // variant "Variant 1".
        }
        with { encoding (field1) "Encoding 2" }
        :
    }
    with { variant "Variant 1" }

    group MyVariantsTwo
    {   :
        type record MyPDUtwo
        {
            integer     field1, // field1 will be encoded according to "Encoding 3"
                                // using encoding variant "Variant 3"
            Mytype      field3  // field3 will be encoded according to "Encoding 3"
                                // using encoding variant "Variant 2"
        }
        with { variant (field1) "Variant 3" }
        :
    }
    with { encode "Encoding 3"; variant "Variant 2"}

}
with { encode "Encoding 1" }
```

27.1.3 Changing attributes of imported language elements

In general, a language element is imported together with its attributes. In some cases these attributes may have to be changed when importing the language element, e.g. a type may be displayed in one module as ASP, then it is imported by another module where it should be displayed as PDU. For such cases it is allowed to change attributes on the **import** statement.

EXAMPLE:

```
import from MyModule {
    type MyType
}
with { display "ASP" }      // MyType will be displayed as ASP
```

```
import from MyModule {
    group MyGroup
}
with {
    display "PDU";          // By default all types will be displayed as PDU
    extension "MyRule"
}
```

27.2 The With statement

The with statement is used to associate attributes to TTCN-3 language elements (and sets thereof).

Syntactical Structure

```
with  "{"
  { ( encode | variant | display | extension | optional )
    [ override ]
    ["(" DefinitionRef | FieldReference | AllRef ")"]
    FreeText [";"] }
"}"
```

Semantic Description

There are five kinds of attributes that can be associated to language elements:

a) **`display:`** allows the specification of display attributes related to specific presentation formats;

b) **`encode:`** allows references to specific encoding rules;

c) **`variant:`** allows references to specific encoding variants;

d) **`extension:`** allows the specification of user-defined attributes;

e) **`optional:`** allows the implicit setting of optional fields in records and sets to omit.

The syntax for the argument of the **`with`** statement (i.e. the actual attributes) is defined as a free text string.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) *DefinitionRef* and *FieldReference* must refer to a definition or field respectively which is within the module, group or definition to which the with statement is associated.

Examples

```
type record MyService {
    integer i,
    float f
}
with { display "ServiceCall" }      // MyRecord will be displayed as a ServiceCall
```

27.3 Display attributes

Display attributes allow the specification of display attributes related to specific presentation formats.

Syntactical Structure

```
display
```

Semantic Description

All TTCN-3 language elements can have **`display`** attributes to specify how particular language elements shall be displayed in, for example, a tabular format.

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

Special attribute strings related to the display attributes for the graphical presentation format can be found in ES 201 873-3 [i.2].

Other **display** attributes may be defined by the user.

NOTE: Because user-defined attributes are not standardized, the interpretation of these attributes may differ between tools or even may not be supported.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

```
type record MyService {
    integer i,
    float f
}
with { display "ServiceCall" }      // MyRecord will be displayed as a ServiceCall
```

27.4 Encoding attributes

In TTCN-3, general or particular encoding rules can be specified by using **encode** and **variant** attributes. Encoding attributes allow references to specific encoding rules.

Syntactical Structure

```
encode
```

Semantic Description

Encoding rules define how a particular value, template, etc. shall be encoded and transmitted over a communication **port** and how received signals shall be decoded. TTCN-3 does not have a default encoding mechanism. This means that encoding rules or encoding directives are defined in some external manner to TTCN-3.

The **encode** attribute allows the association of some referenced encoding rule or encoding directive to be made to a TTCN-3 definition.

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

In most cases encoding attributes will be used in a hierarchical manner. The top-level is the entire module, the next level is a group and the lowest is an individual type or definition:

a) **module:** encoding applies to all types defined in the module, including TTCN-3 types (built-in types);

b) **group:** encoding applies to a group of user-defined type definitions;

c) **type or definition:** encoding applies to a single user-defined type or definition;

d) **field:**encoding applies to a field in a **record** or **set** type or **template**.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

```
module MyFirstmodule
{   :
    import from MySecondModule {
        type MyRecord
    }
    with { encode "MyRule 1" } // Instances of MyRecord will be encoded according to MyRule 1

    :
    type charstring MyType; // Normally encoded according to the 'Global encoding rule
    :
```

```
    group MyRecords
    {   :
        type record MyPDU1
        {
            integer     field1,     // field1 will be encoded according to "Rule 3"
            boolean     field2,     // field2 will be encoded according to "Rule 3"
            Mytype      field3      // field3 will be encoded according to "Rule 2"
        }
        with { encode (field1, field2) "Rule 3" }
        :
    }
    with { encode "Rule 2" }

}
with { encode "Global encoding rule" }
```

27.5 Variant attributes

In TTCN-3, general or particular encoding rules can be specified by using **encode** and **variant** attributes. Variant attributes allow references to specific encoding variants.

Syntactical Structure

```
variant
```

Semantic Description

To specify a refinement of the currently specified encoding scheme instead of its replacement, the **variant** attribute shall be used. The variant attributes are different from other attributes, because they are closely related to encode attributes. Therefore, for variant attributes, additional overwriting rules apply (see clause 27.1.2.1).

Special variant strings:

The following strings are the predefined (standardized) **variant** attributes for simple basic types (see clause D.2.1):

a) "`8 bit`" and "`unsigned 8 bit`" mean, when applied to integer and enumerated types, that the integer value or the integer numbers associated with enumerations shall be handled as it was represented on 8-bits (single byte) within the system.

b) "`16 bit`" and "`unsigned 16 bit`" mean, when applied to integer and enumerated types, that the integer value or the integer numbers associated with enumerations shall be handled as it was represented on 16-bits (two bytes) within the system.

c) "`32 bit`" and "`unsigned 32 bit`" mean, when applied to integer and enumerated types, that the integer value or the integer numbers associated with enumerations shall be handled as it was represented on 32-bits (four bytes) within the system.

d) "`64 bit`" and "`unsigned 64 bit`" mean, when applied to integer and enumerated types, that the integer value or the integer numbers associated with enumerations shall be handled as it was represented on 64-bits (eight bytes) within the system.

e) "`IEEE754 float`","`IEEE754 double`", "`IEEE754 extended float`" and "`IEEE754 extended double`" mean, when applied to a float type, that the value shall be encoded and decoded according to the standard IEEE 754 [6] (see annex E).

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

a) "`UTF-8`" means, when applied to the universal charstring type, that each character of the value shall be individually encoded and decoded according to the UCS Transformation Format 8 (UTF-8) as defined in annex R of ISO/IEC 10646 [2].

b) "`UCS-2`" means, when applied to the universal charstring type, that each character of the value shall be individually encoded and decoded according to the UCS-2 coded representation form (see clause 14.1 of ISO/IEC 10646 [2]).

c) "UTF-16" means, when applied to the universal charstring type, that each character of the value shall be individually encoded and decoded according to the UCS Transformation Format 16 (UTF-16) as defined in annex Q of ISO/IEC 10646 [2].

d) "8 bit" means, when applied to charstring and universal charstring types, that each character of the value shall be individually encoded and decoded according to the coded representation as specified in ISO/IEC 10646 [2] (an 8-bit coding).

The following strings are the predefined (standardized) **variant** attributes for structured types (see clause D.2.3):

a) "IDL:fixed FORMAL/01-12-01 v.2.6" means, when applied to a record type, that the value shall be handled as an IDL fixed point decimal value (see annex E).

These variant attributes can be used in combination with the more general encode attributes specified at a higher level. For example a **universal charstring** specified with the **variant** attribute "UTF-8" within a module which itself has a global encoding attribute "BER:1997" (see clause 12.2 of ES 201 873-7 [i.5]) will cause each character of the values within the string to first be encoded following the UTF-8 rules and then this UTF-8 value will be encoded following the more global BER rules.

Invalid encodings

If it is desired to specify invalid encoding rules then these shall be specified in a referenceable source external to the module in the same way that valid encoding rules are referenced.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

EXAMPLE:

```
module MyTTCNmodule1
{   :
    type charstring MyType; // Normally encoded according to the "Global encoding rule"
    :
    group MyRecords
    {   :
        type record MyPDU1
        {
            integer     field1,     // field1 will be encoded according to "Rule 2"
                                    // using encoding variant "length form 3"
            Mytype      field3      // field3 will be encoded according to "Rule 2"
                                    // using any possible length encoding format
        }
        with { variant (field1) "length form 3" }
        :
    }
    with { encode "Rule 2" }

}
with { encode "Global encoding rule" }
```

27.6 Extension attributes

Extension attributes can be used for proprietary extensions to TTCN-3.

Syntactical Structure

```
extension
```

Semantic Description

All TTCN-3 language elements can have **extension** attributes specified by the user.

NOTE: Because user-defined attributes are not standardized the interpretation of these attributes between tools supplied by different vendors may differ or even not be supported.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5.

Examples

```
testcase MyTestcase() runs on MTCType {
:
}
with { extension "Test Purpose: This test case is used to check …" }
```

27.7 Optional attributes

The **optional** attribute can be used to indicate that optional fields of constants, module parameters or templates of record and set types are implicitly set to **omit**.

Syntactical Structure

```
optional
```

Semantic Description

TTCN-3 constants, module parameters, and templates can have an **optional** attribute. Also, TTCN-3 language elements that contain such definitions, i.e. module, group, function, altstep, test case, control, and component type definitions can have an **optional** attribute. When an **optional** attribute is associated to a function, altstep, test case, control or component type definitions, it shall have effect on all the constants, module parameters, and templates declared within these definitions and not on the enframing definition itself.

Special optional strings:

The following strings are the predefined (standardized) **optional** attributes.

a) "implicit omit" means that all optional fields, which have not been defined in the definition the attribute is associated with, are set to omit. This applies recursively to the optional fields of the entity and to subfields of the mandatory fields.

b) "explicit omit" means that all optional fields, which have not been defined in the definition the attribute is associated with, are left undefined. This applies recursively to the optional fields of the entity and to subfields of the mandatory fields.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5, the following restrictions apply:

a) Data type, port type, procedure signature and variable definitions and import statements shall not have an **optional** attribute associated to them directly. When an **optional** attribute is associated to module, group, function, altstep, test case, control or component type containing such definitions, it shall not have any effect on the included data type, port type, procedure signature, variable or import statement.

Examples

```
type record MyRecord1 {
  integer a,
  boolean b optional
}
type record MyRecord2 {
  MyRecord1 m
}

// reference templates with explicitly set fields
template MyRecord1 MyTemplate1 := { a := ?, b := omit }
template MyRecord2 MyTemplate2 := { m := { a := ?, b := omit }}

// reference templates
template MyRecord1 MyTemplate1a := { a := ? } // b is undefined
template MyRecord1 MyTemplate1b := { a := ? } with {optional "explicit omit"} // b is undefined

template MyRecord2 MyTemplate2a := {} // m and its subfields are undefined
```

```
template MyRecord2 MyTemplate2b := { m := { a := ?}}; // m.b is undefined

// templates with attribute

template MyRecord1 MyTemplate11 := { a := ? } with {optional "implicit omit"}
  // same as MyTemplate1, b is set to omit


template MyRecord2 MyTemplate21 := { m := { a := ?}} with {optional "implicit omit"}
  // same as MyTemplate2, by recursive application of the attribute
template MyRecord2 MyTemplate22 := { m := MyTemplate1a } with {optional "implicit omit"}
  // same as MyTemplate2, by recursive application of the attribute



template MyRecord2 MyTemplate23 := {} with {optional "implicit omit"}
  // same as MyTemplate2a, m remains undefined

template MyRecord2 MyTemplate24 := { m := MyTemplate1b } with {optional "implicit omit"}
  // same as MyTemplate2b, the attribute on the lower scope is not overwritten
template MyRecord2 MyTemplate25 := { m := MyTemplate1b }
  with {optional override "implicit omit"}
  // same as MyTemplate2, the attribute on the lower scope is overwritten
```

Annex A (normative): BNF and static semantics

A.1 TTCN-3 BNF

This annex defines the syntax of TTCN-3 using extended BNF (henceforth just called BNF).

A.1.1 Conventions for the syntax description

Table A.1 defines the metanotation used to specify the extended BNF grammar for TTCN-3.

Table A.1: The syntactic metanotation

::=	is defined to be	definition of non-terminal
abc xyz	abc followed by xyz	concatenation
|	alternative	alternative
[abc]	0 or 1 instances of abc	optional
{abc}	0 or more instances of abc	repetition 1
{abc}+	1 or more instances of abc	repetition 2
(...)	textual grouping	grouping
Abc	the non-terminal symbol abc	non-terminal
"abc"	a terminal symbol abc	terminal

NOTE: The metanotation defined in table A.1 is parsed from left to right. The metanotation operators have the following precedence, from highest (binding tightest) at the top, to lowest (loosest) at the bottom:
- Repetition, Optional
- Grouping
- Concatenation
- Alternative
- Definition.

A.1.2 Statement terminator symbols

In general all TTCN-3 language constructs (i.e. definitions, declarations, statements and operations) are terminated with a semi-colon (;). The semi-colon is optional if the language construct ends with a right-hand curly brace (}) or the following symbol is a right-hand curly brace (}), i.e. the language construct is the last statement in a statement block.

A.1.3 Identifiers

TTCN-3 identifiers are case sensitive and may only contain lowercase letters (a-z) uppercase letters (A-Z) and numeric digits (0-9). Use of the underscore (_) symbol is also allowed. An identifier shall begin with a letter (i.e. not with a number and not an underscore).

A.1.4 Comments

Comments written in free text may appear anywhere in a TTCN-3 specification. Comments may contain any graphical character defined in ISO/IEC 10646 [2]. Block comments shall be opened by the symbol pair /* and closed by the symbol pair */.

EXAMPLE 1:

```
/* This is a block comment
spread over two lines */
```

Block comments shall not be nested.

```
/* This is not /* a legal */ comment */
```

Line comments shall be opened by the symbol pair // and closed by a <*newline*>.

EXAMPLE 2:

```
// This is a line comment
// spread over two lines
```

EXAMPLE 3:

```
// The following is not legal
const // This is MyConst integer MyConst := 1;
// A block comment should have been used instead
const /* This is MyConst */ integer MyConst := 1;
// A line comment like this works as well
const // This is MyConst
    integer MyConst := 1;
```

A.1.5 TTCN-3 terminals

TTCN-3 terminal symbols and reserved words are listed in tables A.2 and A.3.

Table A.2: List of TTCN-3 special terminal symbols

Begin/end block symbols	**{ }**
Begin/end list symbols	**( )**
Alternative symbols	**[ ]**
To symbol (in a range)	**..**
Line comments and Block comments	**/* */ //**
Line/statement terminator symbol	**;**
Arithmetic operator symbols	**+ / -**
Concatenation operator symbol	**&**
Equivalence operator symbols	**!= == >= <=**
String enclosure symbols	**" '**
Wildcard/matching symbols	**? ***
Assignment symbol	**:=**
Communication operation assignment	**->**
Bitstring, hexstring and Octetstring values	**B H O**
Float exponent	**E**

The predefined function identifiers defined in table 15 and described in annex C shall also be treated as reserved words.

Table A.3: List of TTCN-3 terminals which are reserved words

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

The TTCN-3 terminals listed in table A.3 shall not be used as identifiers in a TTCN-3 module. These terminals shall be written in all lowercase letters.

A.1.5.1 Use of whitespaces and newlines

The elements of the TTCN-3 syntax (reserved words, identifiers, terminal symbols and literal values) shall be separated by whitespace or by special terminal symbols listed in table A.2 according to the TTCN-3 syntax.

In representing whitespace, any one or more of the following characters of the C0 set of ITU-T Recommendation T.50 [4] and of annex A of ITU-T Recommendation T.50 [4] may be used in any combination:

- HT - HORIZONTAL TABULATION (9)
- LF - LINE FEED (10)
- VT -VERTICAL TABULATION (11)
- FF - FORM FEED (12)
- CR - CARRIAGE RETURN (13)
- SP - SPACE (32)

The characters of the C0 set of ITU-T Recommendation T.50 [4] and of annex A of ITU-T Recommendation T.50 [4] below are denoting newline (end of line). A single CR(13) character directly followed by an LF(10) character denote a single end of line (i.e. the sequence CRLFCRLFVT denotes 3 lines):

- LF - LINE FEED (10)
- VT - VERTICAL TABULATION (11)
- FF - FORM FEED (12)
- CR - CARRIAGE RETURN (13)

Any character or character sequence that is a valid newline is also a valid whitespace.

NOTE: It is recommended that for newline only the CR and LF and for whitespace only the HT, LF, CR and SP control characters are used as the VT and FF characters may cause problems with some conventional text editors.

A.1.6 TTCN-3 syntax BNF productions

A.1.6.0 TTCN-3 module

```
1.TTCN3Module ::= TTCN3ModuleKeyword ModuleId "{" [ModuleDefinitionsList]
                  [ModuleControlPart] "}" [WithStatement] [SemiColon]
2.TTCN3ModuleKeyword ::= "module"
3.ModuleId ::= Identifier [LanguageSpec]
4.LanguageSpec ::= LanguageKeyword FreeText {"," FreeText}
5.LanguageKeyword ::= "language"
```

A.1.6.1 Module definitions part

A.1.6.1.0 General

```
6.ModuleDefinitionsList ::= {ModuleDefinition [SemiColon]}+
7.ModuleDefinition ::= (([Visibility] (TypeDef |
                                       ConstDef |
                                       TemplateDef |
                                       ModuleParDef |
                                       FunctionDef |
                                       SignatureDef |
                                       TestcaseDef |
                                       AltstepDef |
                                       ImportDef |
                                       ExtFunctionDef |
                                       ExtConstDef
                                      )) |
                        (["public"] GroupDef) |
                        (["private"] FriendModuleDef)
                       ) [WithStatement]
8.Visibility ::= "public" |
                 "friend" |
                 "private"
```

A.1.6.1.1 Typedef definitions

```
9.TypeDef ::= TypeDefKeyword TypeDefBody
10.TypeDefBody ::= StructuredTypeDef | SubTypeDef
11.TypeDefKeyword ::= "type"
12.StructuredTypeDef ::= RecordDef |
                         UnionDef |
                         SetDef |
                         RecordOfDef |
                         SetOfDef |
                         EnumDef |
                         PortDef |
                         ComponentDef
13.RecordDef ::= RecordKeyword Identifier
14.RecordKeyword ::= "record"
15.StructDefBody ::= (Identifier | AddressKeyword) "{" [StructFieldDef
                                                        {"," StructFieldDef}]
                     "}"
16.StructFieldDef ::= (Type | NestedTypeDef) Identifier [ArrayDef] [SubTypeSpec]
                      [OptionalKeyword]
17.NestedTypeDef ::= NestedRecordDef |
                     NestedUnionDef |
                     NestedSetDef |
                     NestedRecordOfDef |
                     NestedSetOfDef |
                     NestedEnumDef
18.NestedRecordDef ::= RecordKeyword "{" [StructFieldDef {"," StructFieldDef}]
                       "}"
19.NestedUnionDef ::= UnionKeyword "{" UnionFieldDef {"," UnionFieldDef}
                      "}"
20.NestedSetDef ::= SetKeyword "{" [StructFieldDef {"," StructFieldDef}]
                    "}"
21.NestedRecordOfDef ::= RecordKeyword [StringLength] OfKeyword (Type |
                                                                 NestedTypeDef)
22.NestedSetOfDef ::= SetKeyword [StringLength] OfKeyword (Type | NestedTypeDef)
23.NestedEnumDef ::= EnumKeyword "{" EnumerationList "}"
24.OptionalKeyword ::= "optional"
25.UnionDef ::= UnionKeyword UnionDefBody
26.UnionKeyword ::= "union"
```

```
27.UnionDefBody ::= (Identifier | AddressKeyword) "{" UnionFieldDef {","
                                                                  UnionFieldDef}
                    "}"
28.UnionFieldDef ::= (Type | NestedTypeDef) Identifier [ArrayDef] [SubTypeSpec]
29.SetDef ::= SetKeyword StructDefBody
30.SetKeyword ::= "set"
31.RecordOfDef ::= RecordKeyword [StringLength] OfKeyword StructOfDefBody
32.OfKeyword ::= "of"
33.StructOfDefBody ::= (Type | NestedTypeDef) (Identifier | AddressKeyword)
                       [SubTypeSpec]
34.SetOfDef ::= SetKeyword [StringLength] OfKeyword StructOfDefBody
35.EnumDef ::= EnumKeyword (Identifier | AddressKeyword) "{" EnumerationList
                "}"
36.EnumKeyword ::= "enumerated"
37.EnumerationList ::= Enumeration {"," Enumeration}
38.Enumeration ::= Identifier ["(" [Minus] Number ")"]
39.SubTypeDef ::= Type (Identifier | AddressKeyword) [ArrayDef] [SubTypeSpec]
40.SubTypeSpec ::= AllowedValues [StringLength] | StringLength

/* STATIC SEMANTICS - AllowedValues shall be of the same type as the field being subtyped */
41.AllowedValues ::= "(" (ValueOrRange {"," ValueOrRange}) | CharStringMatch
                     ")"
42.ValueOrRange ::= RangeDef |
                    ConstantExpression |
                    Type

/* STATIC SEMANTICS - RangeDef production shall only be used with integer, charstring, universal
charstring or float based types */
/* STATIC SEMANTICS - When subtyping charstring or universal charstring range and values shall not
be mixed in the same SubTypeSpec */
43.RangeDef ::= Bound ".." Bound
44.StringLength ::= LengthKeyword "(" SingleExpression [".." Bound] ")"

/* STATIC SEMANTICS - StringLength shall only be used with String types or to limit set of and
record of. SingleExpression and Bound shall evaluate to non-negative integer values (in case of
Bound including infinity) */
45.LengthKeyword ::= "length"
46.PortDef ::= PortKeyword PortDefBody
47.PortDefBody ::= Identifier PortDefAttribs
48.PortKeyword ::= "port"
49.PortDefAttribs ::= MessageAttribs |
                      ProcedureAttribs |
                      MixedAttribs
50.MessageAttribs ::= MessageKeyword "{" [AddressDecl] {MessageList [SemiColon]}+
                      "}" [MapKeyword ParamKeyword "(" FormalValuePar
                           {"," FormalValuePar} ")"] [UnmapKeyword ParamKeyword
                                                      "(" FormalValuePar
                                                      {"," FormalValuePar}
                                                      ")"]
51.AddressDecl ::= AddressKeyword Type SemiColon
52.MessageList ::= Direction AllOrTypeList
53.Direction ::= InParKeyword |
                 OutParKeyword |
                 InOutParKeyword
54.MessageKeyword ::= "message"
55.AllOrTypeList ::= AllKeyword | TypeList

/* NOTE: The use of AllKeyword in port definitions is deprecated */
56.AllKeyword ::= "all"
57.TypeList ::= Type {"," Type}
58.ProcedureAttribs ::= ProcedureKeyword "{" [AddressDecl] {ProcedureList
                                                            [SemiColon]}+
                        "}" [MapKeyword ParamKeyword "(" FormalValuePar
                             {"," FormalValuePar} ")"] [UnmapKeyword ParamKeyword
                                                        "(" FormalValuePar
                                                        {"," FormalValuePar}
                                                        ")"]
59.ProcedureKeyword ::= "procedure"
60.ProcedureList ::= Direction AllOrSignatureList
61.AllOrSignatureList ::= AllKeyword | SignatureList
62.SignatureList ::= Signature {"," Signature}
63.MixedAttribs ::= MixedKeyword "{" [AddressDecl] {MixedList [SemiColon]}+
                    "}"
64.MixedKeyword ::= "mixed"
65.MixedList ::= Direction ProcOrTypeList
66.ProcOrTypeList ::= AllKeyword | (ProcOrType {"," ProcOrType})
67.ProcOrType ::= Signature | Type
68.ComponentDef ::= ComponentKeyword Identifier [ExtendsKeyword ComponentType
```

```
                                                {"," ComponentType}] "{"
                    [ComponentDefList] "}"
69.ComponentKeyword ::= "component"
70.ExtendsKeyword ::= "extends"
71.ComponentType ::= ExtendedIdentifier
72.ComponentDefList ::= {ComponentElementDef [SemiColon]}
73.ComponentElementDef ::= PortInstance |
                           VarInstance |
                           TimerInstance |
                           ConstDef
74.PortInstance ::= PortKeyword ExtendedIdentifier PortElement {"," PortElement}
75.PortElement ::= Identifier [ArrayDef]
```

A.1.6.1.2 Constant definitions

```
76.ConstDef ::= ConstKeyword Type ConstList
77.ConstList ::= SingleConstDef {"," SingleConstDef}
78.SingleConstDef ::= Identifier [ArrayDef] AssignmentChar ConstantExpression
79.ConstKeyword ::= "const"
```

A.1.6.1.3 Template definitions

```
80.TemplateDef ::= TemplateKeyword [TemplateRestriction] BaseTemplate
                   [DerivedDef] AssignmentChar TemplateBody
81.BaseTemplate ::= (Type | Signature) Identifier ["(" TemplateOrValueFormalParList
                                                  ")"]
82.TemplateKeyword ::= "template"
83.DerivedDef ::= ModifiesKeyword ExtendedIdentifier
84.ModifiesKeyword ::= "modifies"
85.TemplateOrValueFormalParList ::= TemplateOrValueFormalPar {"," TemplateOrValueFormalPar}
86.TemplateOrValueFormalPar ::= FormalValuePar | FormalTemplatePar

/* STATIC SEMANTICS - FormalValuePar shall resolve to an in parameter */
87.TemplateBody ::= (SimpleSpec |
                     FieldSpecList |
                     ArrayValueOrAttrib
                    ) [ExtraMatchingAttributes]

/* STATIC SEMANTICS - Within TeplateBody the ArrayValueOrAttrib can be used for array, record,
record of and set of types. */
88.SimpleSpec ::= (SingleExpression ["&" SimpleTemplateSpec]) | SimpleTemplateSpec
89.SimpleTemplateSpec ::= CharStringMatch | (SingleTemplateExpression
                                             ["&" SimpleSpec])
90.SingleTemplateExpression ::= MatchingSymbol | (TemplateRefWithParList
                                                  [ExtendedFieldReference])
91.FieldSpecList ::= "{" [FieldSpec {"," FieldSpec}] "}"
92.FieldSpec ::= FieldReference AssignmentChar TemplateBody
93.FieldReference ::= StructFieldRef |
                      ArrayOrBitRef |
                      ParRef
94.StructFieldRef ::= Identifier |
                      PredefinedType |
                      TypeReference

/* STATIC SEMANTICS - PredefinedType and TypeReference shall be used for anytype value notation
only. PredefinedType shall not be AnyTypeKeyword.*/
95.ParRef ::= Identifier

/* STATIC SEMANTICS - Identifier in ParRef shall be a formal parameter identifier from the
associated signature definition */
96.ArrayOrBitRef ::= "[" FieldOrBitNumber "]"

/* STATIC SEMANTICS - ArrayRef shall be optionally used for array types and TTCN-3 record of and set
of. The same notation can be used for a Bit reference inside an TTCN-3 charstring, universal
charstring, bitstring, octetstring and hexstring type */
97.FieldOrBitNumber ::= SingleExpression

/* STATIC SEMANTICS - SingleExpression will resolve to a value of integer type */
98.ArrayValueOrAttrib ::= "{" ArrayElementSpecList "}"
99.ArrayElementSpecList ::= ArrayElementSpec {"," ArrayElementSpec}
100.ArrayElementSpec ::= Minus |
                         PermutationMatch |
                         TemplateBody
101.MatchingSymbol ::= Complement |
                       (AnyValue [WildcardLengthMatch]) |
                       (AnyOrOmit [WildcardLengthMatch]) |
                       ValueOrAttribList |
                       Range |
```

```
                        BitStringMatch |
                        HexStringMatch |
                        OctetStringMatch |
                        CharStringMatch |
                        SubsetMatch |
                        SupersetMatch

/* STATIC SEMANTIC - WildcardLengthMatch must be used when MatchingSymbol is used in fractions of a
concatenated string or list (see clause 15.11) and shall not be used in other cases. In this case,
the Complement, ValueOrAttribList, Range, BitStringMatch, HexStringMatch, OctetStringMatch,
CharStringMatch, SubsetMatch and SupersetMatch productions shall not be used. */
102.ExtraMatchingAttributes ::= StringLength |
                                IfPresentKeyword |
                                (StringLength IfPresentKeyword)
103.BitStringMatch ::= "'" {BinOrMatch} "'" "B"
104.BinOrMatch ::= Bin |
                   AnyValue |
                   AnyOrOmit
105.HexStringMatch ::= "'" {HexOrMatch} "'" "H"
106.HexOrMatch ::= Hex |
                   AnyValue |
                   AnyOrOmit
107.OctetStringMatch ::= "'" {OctOrMatch} "'" "O"
108.OctOrMatch ::= Oct |
                   AnyValue |
                   AnyOrOmit
109.CharStringMatch ::= PatternKeyword PatternParticle {"&" PatternParticle}
110.PatternParticle ::= Pattern | ReferencedValue
111.PatternKeyword ::= "pattern"
112.Pattern ::= """ {PatternElement} """
113.PatternElement ::= ("\" ("?" | "*" | "\" | "[" | "]" | "{" | "}" |
                           """ | "|" | "(" | ")" | "#" | "+" | "d" |
                           "w" | "t" | "n" | "r" | "s" | "b"
                          )) | ("?" | "*" | "\" | "|" | "+"
                               ) | ("[" ["^"] [{PatternChar ["-" PatternChar]}]
                                    "]") |
                      ("{" ReferencedValue "}") |
                      ("\" "N" "{" (ReferencedValue | Type) "}") |
                      (""" """) |
                      ("(" PatternElement ")") |
                      ("#" (Num |
                            ("(" Num "," [Num] ")") | ("(" "," Num ")")
                           )) |
                      PatternChar
114.PatternChar ::= Char | PatternQuadruple
115.PatternQuadruple ::= "\" "q" "(" Number "," Number "," Number ","
                         Number ")"
116.Complement ::= ComplementKeyword "(" TemplateBody {"," TemplateBody}
                   ")"
117.ComplementKeyword ::= "complement"
118.SubsetMatch ::= SubsetKeyword ValueOrAttribList
119.SubsetKeyword ::= "subset"
120.SupersetMatch ::= SupersetKeyword ValueOrAttribList
121.SupersetKeyword ::= "superset"
122.PermutationMatch ::= PermutationKeyword PermutationList
123.PermutationKeyword ::= "permutation"
124.PermutationList ::= "(" TemplateBody {"," TemplateBody} ")"

/* STATIC SEMANTICS: Restrictions on the content of TemplateBody are given in clause B.1.3.3. */
125.AnyValue ::= "?"
126.AnyOrOmit ::= "*"
127.ValueOrAttribList ::= "(" TemplateBody {"," TemplateBody}+ ")"
128.WildcardLengthMatch ::= LengthKeyword "(" ConstantExpression ")"
129.IfPresentKeyword ::= "ifpresent"
130.PresentKeyword ::= "present"
131.Range ::= "(" Bound ".." Bound ")"
132.Bound ::= (["!"] SingleExpression) | ([Minus] InfinityKeyword)

/* STATIC SEMANTICS - Bounds shall evaluate to types integer, charstring, universal charstring or
float. In case they evaluate to types charstring or universal charstring, the string length shall be
1. infinity as lower bound and -infinity as upper bound are allowed for float types only. */
133.InfinityKeyword ::= "infinity"
134.TemplateInstanceAssignment ::= Identifier ":=" InLineTemplate

/* STATIC SEMANTICS - if a value parameter is used, the inline template shall evaluate to a value */
135.TemplateRefWithParList ::= ExtendedIdentifier [TemplateActualParList]
136.InLineTemplate ::= [(Type | Signature) Colon] [DerivedRefWithParList
                                                   AssignmentChar] TemplateBody
```

```
137.DerivedRefWithParList ::= ModifiesKeyword TemplateRefWithParList
138.TemplateActualParList ::= "(" [(TemplateInstanceActualPar {"," TemplateInstanceActualPar}) |
                                  (TemplateInstanceAssignment {"," TemplateInstanceAssignment})]
                              ")"
139.TemplateInstanceActualPar ::= InLineTemplate | Minus

/* STATIC SEMANTICS - When the corresponding formal parameter is not of template type the
TemplateInstance production shall resolve to one or more SingleExpressions */
140.TemplateOps ::= MatchOp | ValueofOp
141.MatchOp ::= MatchKeyword "(" Expression "," InLineTemplate ")"
142.MatchKeyword ::= "match"
143.ValueofOp ::= ValueofKeyword "(" InLineTemplate ")"
144.ValueofKeyword ::= "valueof"
```

A.1.6.1.4 Function definitions

```
145.FunctionDef ::= FunctionKeyword Identifier "(" [FunctionFormalParList]
                    ")" [RunsOnSpec] [ReturnType] StatementBlock
146.FunctionKeyword ::= "function"
147.FunctionFormalParList ::= FunctionFormalPar {"," FunctionFormalPar}
148.FunctionFormalPar ::= FormalValuePar |
                          FormalTimerPar |
                          FormalTemplatePar |
                          FormalPortPar
149.ReturnType ::= ReturnKeyword [TemplateKeyword | RestrictedTemplate]
                   Type
150.ReturnKeyword ::= "return"
151.RunsOnSpec ::= RunsKeyword OnKeyword ComponentType
152.RunsKeyword ::= "runs"
153.OnKeyword ::= "on"
154.MTCKeyword ::= "mtc"
155.StatementBlock ::= "{" [FunctionDefList] [FunctionStatementList] "}"
156.FunctionDefList ::= {(FunctionLocalDef | FunctionLocalInst) [SemiColon]}+
157.FunctionStatementList ::= {FunctionStatement [SemiColon]}+
158.FunctionLocalInst ::= VarInstance | TimerInstance
159.FunctionLocalDef ::= ConstDef | TemplateDef
160.FunctionStatement ::= ConfigurationStatements |
                          TimerStatements |
                          CommunicationStatements |
                          BasicStatements |
                          BehaviourStatements |
                          SetLocalVerdict |
                          SUTStatements |
                          TestcaseOperation
161.FunctionInstance ::= FunctionRef "(" [FunctionActualParList] ")"
162.FunctionRef ::= [Identifier Dot] (Identifier | PreDefFunctionIdentifier)
163.PreDefFunctionIdentifier ::= Identifier

/* STATIC SEMANTICS - The Identifier shall be one of the pre-defined TTCN-3 Function Identifiers
from Annex C of ES 201 873-1 */
164.FunctionActualParList ::= (FunctionActualPar {"," FunctionActualPar}) |
                              (FunctionActualParAssignment {"," FunctionActualParAssignment})
165.FunctionActualPar ::= ArrayIdentifierRef |
                          InLineTemplate |
                          ComponentRef |
                          Minus

/* STATIC SEMANTICS - When the corresponding formal parameter is not of template type the
TemplateInstance production shall resolve to one or more SingleExpressions i.e. equivalent to the
Expression production */
166.FunctionActualParAssignment ::= TemplateInstanceAssignment |
                                    ComponentRefAssignment |
                                    ArrayIdentifierRefAssignment
167.ArrayIdentifierRefAssignment ::= Identifier ":=" ArrayIdentifierRef
```

A.1.6.1.5 Signature definitions

```
168.SignatureDef ::= SignatureKeyword Identifier "(" [SignatureFormalParList]
                     ")" [ReturnType | NoBlockKeyword] [ExceptionSpec]
169.SignatureKeyword ::= "signature"
170.SignatureFormalParList ::= FormalValuePar {"," FormalValuePar}
171.ExceptionSpec ::= ExceptionKeyword "(" TypeList ")"
172.ExceptionKeyword ::= "exception"
173.Signature ::= ExtendedIdentifier
174.NoBlockKeyword ::= "noblock"
```

A.1.6.1.6 Testcase definitions

```
175.TestcaseDef ::= TestcaseKeyword Identifier "(" [TemplateOrValueFormalParList]
                    ")" ConfigSpec StatementBlock
176.TestcaseKeyword ::= "testcase"
177.ConfigSpec ::= RunsOnSpec [SystemSpec]
178.SystemSpec ::= SystemKeyword ComponentType
179.SystemKeyword ::= "system"
180.TestcaseInstance ::= ExecuteKeyword "(" ExtendedIdentifier "(" [TestcaseActualParList]
                         ")" ["," (Expression | Minus) ["," SingleExpression]]
                         ")"
181.ExecuteKeyword ::= "execute"
182.TestcaseActualParList ::= (TemplateInstanceActualPar {"," TemplateInstanceActualPar}) |
                              (TemplateInstanceAssignment {"," TemplateInstanceAssignment})

/* STATIC SEMANTICS - When the corresponding formal parameter is not of template type the
TemplateInstance production shall resolve to one or more SingleExpressions i.e. equivalent to the
Expression production */
```

A.1.6.1.7 Altstep definitions

```
183.AltstepDef ::= AltstepKeyword Identifier "(" [FunctionFormalParList]
                   ")" [RunsOnSpec] "{" AltstepLocalDefList AltGuardList
                   "}"
184.AltstepKeyword ::= "altstep"
185.AltstepLocalDefList ::= {AltstepLocalDef [SemiColon]}
186.AltstepLocalDef ::= VarInstance |
                        TimerInstance |
                        ConstDef |
                        TemplateDef
187.AltstepInstance ::= ExtendedIdentifier "(" [FunctionActualParList]
                        ")"
```

A.1.6.1.8 Import definitions

```
188.ImportDef ::= ImportKeyword ImportFromSpec (AllWithExcepts | ("{"
                                                                  ImportSpec
                                                                  "}"))
189.ImportKeyword ::= "import"
190.AllWithExcepts ::= AllKeyword [ExceptsDef]
191.ExceptsDef ::= ExceptKeyword "{" ExceptSpec "}"
192.ExceptKeyword ::= "except"
193.ExceptSpec ::= {ExceptElement [SemiColon]}
194.ExceptElement ::= ExceptGroupSpec |
                      ExceptTypeDefSpec |
                      ExceptTemplateSpec |
                      ExceptConstSpec |
                      ExceptTestcaseSpec |
                      ExceptAltstepSpec |
                      ExceptFunctionSpec |
                      ExceptSignatureSpec |
                      ExceptModuleParSpec
195.ExceptGroupSpec ::= GroupKeyword (FullGroupIdentifierList | AllKeyword)
196.IdentifierListOrAll ::= IdentifierList | AllKeyword
197.ExceptTypeDefSpec ::= TypeDefKeyword IdentifierListOrAll
198.ExceptTemplateSpec ::= TemplateKeyword IdentifierListOrAll
199.ExceptConstSpec ::= ConstKeyword IdentifierListOrAll
200.ExceptTestcaseSpec ::= TestcaseKeyword IdentifierListOrAll
201.ExceptAltstepSpec ::= AltstepKeyword IdentifierListOrAll
202.ExceptFunctionSpec ::= FunctionKeyword IdentifierListOrAll
203.ExceptSignatureSpec ::= SignatureKeyword IdentifierListOrAll
204.ExceptModuleParSpec ::= ModuleParKeyword IdentifierListOrAll
205.ImportSpec ::= {ImportElement [SemiColon]}
206.ImportElement ::= ImportGroupSpec |
                     ImportTypeDefSpec |
                     ImportTemplateSpec |
                     ImportConstSpec |
                     ImportTestcaseSpec |
                     ImportAltstepSpec |
                     ImportFunctionSpec |
                     ImportSignatureSpec |
                     ImportModuleParSpec |
                     ImportImportSpec
207.ImportFromSpec ::= FromKeyword ModuleId [RecursiveKeyword]
208.RecursiveKeyword ::= "recursive"
209.ImportGroupSpec ::= GroupKeyword (GroupRefListWithExcept | AllGroupsWithExcept)
210.GroupRefListWithExcept ::= FullGroupIdentifierWithExcept {"," FullGroupIdentifierWithExcept}
```

```
211.AllGroupsWithExcept ::= AllKeyword [ExceptKeyword FullGroupIdentifierList]
212.FullGroupIdentifier ::= Identifier {Dot Identifier}
213.FullGroupIdentifierWithExcept ::= FullGroupIdentifier [ExceptsDef]
214.IdentifierListOrAllWithExcept ::= IdentifierList | AllWithExcept
215.ImportTypeDefSpec ::= TypeDefKeyword IdentifierListOrAllWithExcept
216.AllWithExcept ::= AllKeyword [ExceptKeyword IdentifierList]
217.ImportTemplateSpec ::= TemplateKeyword IdentifierListOrAllWithExcept
218.ImportConstSpec ::= ConstKeyword IdentifierListOrAllWithExcept
219.ImportAltstepSpec ::= AltstepKeyword IdentifierListOrAllWithExcept
220.ImportTestcaseSpec ::= TestcaseKeyword IdentifierListOrAllWithExcept
221.ImportFunctionSpec ::= FunctionKeyword IdentifierListOrAllWithExcept
222.ImportSignatureSpec ::= SignatureKeyword IdentifierListOrAllWithExcept
223.ImportModuleParSpec ::= ModuleParKeyword IdentifierListOrAllWithExcept
224.ImportImportSpec ::= ImportKeyword AllKeyword
```

A.1.6.1.9 Group definitions

```
225.GroupDef ::= GroupKeyword Identifier "{" [ModuleDefinitionsList] "}"
226.GroupKeyword ::= "group"
```

A.1.6.1.10 External function definitions

```
227.ExtFunctionDef ::= ExtKeyword FunctionKeyword Identifier "(" [FunctionFormalParList]
                       ")" [ReturnType]
228.ExtKeyword ::= "external"
```

A.1.6.1.11 External constant definitions

```
229.ExtConstDef ::= ExtKeyword ConstKeyword Type IdentifierList
```

A.1.6.1.12 Module parameter definitions

```
230.ModuleParDef ::= ModuleParKeyword (ModulePar | ("{" MultitypedModuleParList
                                                    "}"))
231.ModuleParKeyword ::= "modulepar"
232.MultitypedModuleParList ::= {ModulePar [SemiColon]}
233.ModulePar ::= Type ModuleParList
234.ModuleParList ::= Identifier [AssignmentChar ConstantExpression] {","
                                                                      Identifier
                                                                      [AssignmentChar
                                                                      ConstantExpression]}
```

A.1.6.1.13 Friend module definitions

```
235.FriendModuleDef ::= "friend" "module" IdentifierList [SemiColon]
```

A.1.6.2 Control part

```
236.ModuleControlPart ::= ControlKeyword "{" ModuleControlBody "}" [WithStatement]
                          [SemiColon]
237.ControlKeyword ::= "control"
238.ModuleControlBody ::= [ControlStatementOrDefList]
239.ControlStatementOrDefList ::= {ControlStatementOrDef [SemiColon]}+
240.ControlStatementOrDef ::= FunctionLocalDef |
                              FunctionLocalInst |
                              ControlStatement
241.ControlStatement ::= TimerStatements |
                         BasicStatements |
                         BehaviourStatements |
                         SUTStatements |
                         StopKeyword
```

A.1.6.3 Local definitions

A.1.6.3.1 Variable instantiation

```
242.VarInstance ::= VarKeyword ((Type VarList) | ((TemplateKeyword | RestrictedTemplate)
                                                  Type TempVarList))
243.VarList ::= SingleVarInstance {"," SingleVarInstance}
244.SingleVarInstance ::= Identifier [ArrayDef] [AssignmentChar Expression]
245.VarKeyword ::= "var"
246.TempVarList ::= SingleTempVarInstance {"," SingleTempVarInstance}
247.SingleTempVarInstance ::= Identifier [ArrayDef] [AssignmentChar TemplateBody]
248.VariableRef ::= Identifier [ExtendedFieldReference]
```

A.1.6.3.2 Timer instantiation

```
249.TimerInstance ::= TimerKeyword VarList
250.TimerKeyword ::= "timer"
251.ArrayIdentifierRef ::= Identifier {ArrayOrBitRef}
```

A.1.6.4 Operations

A.1.6.4.1 Component operations

```
252.ConfigurationStatements ::= ConnectStatement |
                                MapStatement |
                                DisconnectStatement |
                                UnmapStatement |
                                DoneStatement |
                                KilledStatement |
                                StartTCStatement |
                                StopTCStatement |
                                KillTCStatement
253.ConfigurationOps ::= CreateOp |
                         SelfOp |
                         SystemKeyword |
                         MTCKeyword |
                         RunningOp |
                         AliveOp
254.CreateOp ::= ComponentType Dot CreateKeyword ["(" (SingleExpression |
                                                       Minus) ["," SingleExpression]
                                                    ")"] [AliveKeyword]
255.SelfOp ::= "self"
256.DoneStatement ::= ComponentId Dot DoneKeyword
257.KilledStatement ::= ComponentId Dot KilledKeyword
258.ComponentId ::= ComponentOrDefaultReference | (AnyKeyword | AllKeyword)
                    ComponentKeyword
259.DoneKeyword ::= "done"
260.KilledKeyword ::= "killed"
261.RunningOp ::= ComponentId Dot RunningKeyword
262.RunningKeyword ::= "running"
263.AliveOp ::= ComponentId Dot AliveKeyword
264.CreateKeyword ::= "create"
265.AliveKeyword ::= "alive"
266.ConnectStatement ::= ConnectKeyword SingleConnectionSpec
267.ConnectKeyword ::= "connect"
268.SingleConnectionSpec ::= "(" PortRef "," PortRef ")"
269.PortRef ::= ComponentRef Colon ArrayIdentifierRef
270.ComponentRef ::= ComponentOrDefaultReference |
                   SystemKeyword |
                   SelfOp |
                   MTCKeyword
271.ComponentRefAssignment ::= Identifier ":=" ComponentRef
272.DisconnectStatement ::= DisconnectKeyword [SingleOrMultiConnectionSpec]
273.SingleOrMultiConnectionSpec ::= SingleConnectionSpec |
                                   AllConnectionsSpec |
                                   AllPortsSpec |
                                   AllCompsAllPortsSpec
274.AllConnectionsSpec ::= "(" PortRef ")"
275.AllPortsSpec ::= "(" ComponentRef ":" AllKeyword PortKeyword ")"
276.AllCompsAllPortsSpec ::= "(" AllKeyword ComponentKeyword ":" AllKeyword
                            PortKeyword ")"
277.DisconnectKeyword ::= "disconnect"
278.MapStatement ::= MapKeyword SingleConnectionSpec [ParamKeyword FunctionActualParList]
279.MapKeyword ::= "map"
280.UnmapStatement ::= UnmapKeyword [SingleOrMultiConnectionSpec] [ParamKeyword
                                                                  FunctionActualParList]
281.UnmapKeyword ::= "unmap"
282.StartTCStatement ::= ComponentOrDefaultReference Dot StartKeyword
                        "(" FunctionInstance ")"
283.StartKeyword ::= "start"
284.StopTCStatement ::= StopKeyword | (ComponentReferenceOrLiteral | AllKeyword
                                      ComponentKeyword) Dot StopKeyword
285.ComponentReferenceOrLiteral ::= ComponentOrDefaultReference |
                                   MTCKeyword |
                                   SelfOp
286.KillTCStatement ::= KillKeyword | ((ComponentReferenceOrLiteral |
                                       AllKeyword ComponentKeyword) Dot
                                      KillKeyword)
287.ComponentOrDefaultReference ::= VariableRef | FunctionInstance
```

```
288.KillKeyword ::= "kill"
```

A.1.6.4.2 Port operations

```
289.CommunicationStatements ::= SendStatement |
                                CallStatement |
                                ReplyStatement |
                                RaiseStatement |
                                ReceiveStatement |
                                TriggerStatement |
                                GetCallStatement |
                                GetReplyStatement |
                                CatchStatement |
                                CheckStatement |
                                ClearStatement |
                                StartStatement |
                                StopStatement |
                                HaltStatement
290.SendStatement ::= ArrayIdentifierRef Dot PortSendOp
291.PortSendOp ::= SendOpKeyword "(" InLineTemplate ")" [ToClause]
292.SendOpKeyword ::= "send"
293.ToClause ::= ToKeyword (InLineTemplate |
                            AddressRefList |
                            AllKeyword ComponentKeyword
                           )
294.AddressRefList ::= "(" InLineTemplate {"," InLineTemplate} ")"
295.ToKeyword ::= "to"
296.CallStatement ::= ArrayIdentifierRef Dot PortCallOp [PortCallBody]
297.PortCallOp ::= CallOpKeyword "(" CallParameters ")" [ToClause]
298.CallOpKeyword ::= "call"
299.CallParameters ::= InLineTemplate ["," CallTimerValue]
300.CallTimerValue ::= Expression | NowaitKeyword
301.NowaitKeyword ::= "nowait"
302.PortCallBody ::= "{" CallBodyStatementList "}"
303.CallBodyStatementList ::= {CallBodyStatement [SemiColon]}+
304.CallBodyStatement ::= CallBodyGuard StatementBlock
305.CallBodyGuard ::= AltGuardChar CallBodyOps
306.CallBodyOps ::= GetReplyStatement | CatchStatement
307.ReplyStatement ::= ArrayIdentifierRef Dot PortReplyOp
308.PortReplyOp ::= ReplyKeyword "(" InLineTemplate [ReplyValue] ")" [ToClause]
309.ReplyKeyword ::= "reply"
310.ReplyValue ::= ValueKeyword Expression
311.RaiseStatement ::= ArrayIdentifierRef Dot PortRaiseOp
312.PortRaiseOp ::= RaiseKeyword "(" Signature "," InLineTemplate ")"
                   [ToClause]
313.RaiseKeyword ::= "raise"
314.ReceiveStatement ::= PortOrAny Dot PortReceiveOp
315.PortOrAny ::= ArrayIdentifierRef | AnyKeyword PortKeyword
316.PortReceiveOp ::= ReceiveOpKeyword ["(" InLineTemplate ")"] [FromClause]
                     [PortRedirect]
317.ReceiveOpKeyword ::= "receive"
318.FromClause ::= FromKeyword (InLineTemplate |
                               AddressRefList |
                               AnyKeyword ComponentKeyword
                              )
319.FromKeyword ::= "from"
320.PortRedirect ::= PortRedirectSymbol (ValueSpec [SenderSpec] | SenderSpec)
321.PortRedirectSymbol ::= "->"
322.ValueSpec ::= ValueKeyword (VariableRef | ("(" SingleValueSpec {","
                                                                   SingleValueSpec}
                                              ")"))
323.SingleValueSpec ::= VariableRef [AssignmentChar FieldReference ExtendedFieldReference]

/*STATIC SEMANTICS - FieldReference shall not be ParRef and ExtendedFieldReference shall not be
TypeDefIdentifier*/
324.ValueKeyword ::= "value"
325.SenderSpec ::= SenderKeyword VariableRef
326.SenderKeyword ::= "sender"
327.TriggerStatement ::= PortOrAny Dot PortTriggerOp
328.PortTriggerOp ::= TriggerOpKeyword ["(" InLineTemplate ")"] [FromClause]
                      [PortRedirect]
329.TriggerOpKeyword ::= "trigger"
330.GetCallStatement ::= PortOrAny Dot PortGetCallOp
331.PortGetCallOp ::= GetCallOpKeyword ["(" InLineTemplate ")"] [FromClause]
                     [PortRedirectWithParam]
332.GetCallOpKeyword ::= "getcall"
333.PortRedirectWithParam ::= PortRedirectSymbol RedirectWithParamSpec
334.RedirectWithParamSpec ::= ParamSpec [SenderSpec] | SenderSpec
```

```
335.ParamSpec ::= ParamKeyword ParamAssignmentList
336.ParamKeyword ::= "param"
337.ParamAssignmentList ::= "(" (AssignmentList | VariableList) ")"
338.AssignmentList ::= VariableAssignment {"," VariableAssignment}
339.VariableAssignment ::= VariableRef AssignmentChar Identifier
340.VariableList ::= VariableEntry {"," VariableEntry}
341.VariableEntry ::= VariableRef | Minus
342.GetReplyStatement ::= PortOrAny Dot PortGetReplyOp
343.PortGetReplyOp ::= GetReplyOpKeyword ["(" InLineTemplate [ValueMatchSpec]
                                          ")"] [FromClause] [PortRedirectWithValueAndParam]
344.PortRedirectWithValueAndParam ::= PortRedirectSymbol RedirectWithValueAndParamSpec
345.RedirectWithValueAndParamSpec ::= ValueSpec [ParamSpec] [SenderSpec] |
                                     RedirectWithParamSpec
346.GetReplyOpKeyword ::= "getreply"
347.ValueMatchSpec ::= ValueKeyword InLineTemplate
348.CheckStatement ::= PortOrAny Dot PortCheckOp
349.PortCheckOp ::= CheckOpKeyword ["(" CheckParameter ")"]
350.CheckOpKeyword ::= "check"
351.CheckParameter ::= CheckPortOpsPresent |
                      FromClausePresent |
                      RedirectPresent
352.FromClausePresent ::= FromClause [PortRedirectSymbol SenderSpec]
353.RedirectPresent ::= PortRedirectSymbol SenderSpec
354.CheckPortOpsPresent ::= PortReceiveOp |
                           PortGetCallOp |
                           PortGetReplyOp |
                           PortCatchOp
355.CatchStatement ::= PortOrAny Dot PortCatchOp
356.PortCatchOp ::= CatchOpKeyword ["(" CatchOpParameter ")"] [FromClause]
                    [PortRedirect]
357.CatchOpKeyword ::= "catch"
358.CatchOpParameter ::= Signature "," InLineTemplate | TimeoutKeyword
359.ClearStatement ::= PortOrAll Dot ClearOpKeyword
360.PortOrAll ::= ArrayIdentifierRef | AllKeyword PortKeyword
361.ClearOpKeyword ::= "clear"
362.StartStatement ::= PortOrAll Dot StartKeyword
363.StopStatement ::= PortOrAll Dot StopKeyword
364.StopKeyword ::= "stop"
365.HaltStatement ::= PortOrAll Dot HaltKeyword
366.HaltKeyword ::= "halt"
367.AnyKeyword ::= "any"
```

A.1.6.4.3 Timer operations

```
368.TimerStatements ::= StartTimerStatement |
                        StopTimerStatement |
                        TimeoutStatement
369.TimerOps ::= ReadTimerOp | RunningTimerOp
370.StartTimerStatement ::= ArrayIdentifierRef Dot StartKeyword ["(" Expression
                                                                 ")"]
371.StopTimerStatement ::= TimerRefOrAll Dot StopKeyword
372.TimerRefOrAll ::= ArrayIdentifierRef | AllKeyword TimerKeyword
373.ReadTimerOp ::= ArrayIdentifierRef Dot ReadKeyword
374.ReadKeyword ::= "read"
375.RunningTimerOp ::= TimerRefOrAny Dot RunningKeyword
376.TimeoutStatement ::= TimerRefOrAny Dot TimeoutKeyword
377.TimerRefOrAny ::= ArrayIdentifierRef | (AnyKeyword TimerKeyword)
378.TimeoutKeyword ::= "timeout"
```

A.1.6.4.4 Testcase operation

```
379.TestcaseOperation ::= TestcaseKeyword "." StopKeyword ["(" {(FreeText |
                                                                 InLineTemplate)
                                                                ["," ]}
                                                           ")"]
```

A.1.6.5 Type

```
380.Type ::= PredefinedType | ReferencedType
381.PredefinedType ::= BitStringKeyword |
                       BooleanKeyword |
                       CharStringKeyword |
                       UniversalCharString |
                       IntegerKeyword |
                       OctetStringKeyword |
                       HexStringKeyword |
                       VerdictTypeKeyword |
                       FloatKeyword |
```

```
                        AddressKeyword |
                        DefaultKeyword |
                        AnyTypeKeyword
382.BitStringKeyword ::= "bitstring"
383.BooleanKeyword ::= "boolean"
384.IntegerKeyword ::= "integer"
385.OctetStringKeyword ::= "octetstring"
386.HexStringKeyword ::= "hexstring"
387.VerdictTypeKeyword ::= "verdicttype"
388.FloatKeyword ::= "float"
389.AddressKeyword ::= "address"
390.DefaultKeyword ::= "default"
391.AnyTypeKeyword ::= "anytype"
392.CharStringKeyword ::= "charstring"
393.UniversalCharString ::= UniversalKeyword CharStringKeyword
394.UniversalKeyword ::= "universal"
395.ReferencedType ::= ExtendedIdentifier [ExtendedFieldReference]
396.TypeReference ::= Identifier
397.ArrayDef ::= {"[" SingleExpression [".." SingleExpression] "]"}+

/* STATIC SEMANTICS - ArrayBounds will resolve to a non negative value of integer type */
```

A.1.6.6 Value

```
398.Value ::= PredefinedValue | ReferencedValue
399.PredefinedValue ::= Bstring |
                        BooleanValue |
                        CharStringValue |
                        Number | /* IntegerValue */
                        Ostring |
                        Hstring |
                        VerdictTypeValue |
                        Identifier | /* EnumeratedValue */
                        FloatValue |
                        AddressValue |
                        OmitKeyword
400.BooleanValue ::= "true" | "false"
401.VerdictTypeValue ::= "pass" |
                         "fail" |
                         "inconc" |
                         "none" |
                         "error"
402.CharStringValue ::= Cstring | Quadruple
403.Quadruple ::= CharKeyword "(" Number "," Number "," Number "," Number
                  ")"
404.CharKeyword ::= "char"
405.FloatValue ::= FloatDotNotation |
                   FloatENotation |
                   NaNKeyword
406.NaNKeyword ::= "not_a_number"
407.FloatDotNotation ::= Number Dot DecimalNumber
408.FloatENotation ::= Number [Dot DecimalNumber] Exponential [Minus]
                       Number
409.Exponential ::= "E"
410.ReferencedValue ::= ExtendedIdentifier [ExtendedFieldReference]
411.Number ::= (NonZeroNum {Num}) | "0"
412.NonZeroNum ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
413.DecimalNumber ::= {Num}+
414.Num ::= "0" | NonZeroNum
415.Bstring ::= "'" {Bin} "'" "B"
416.Bin ::= "0" | "1"
417.Hstring ::= "'" {Hex} "'" "H"
418.Hex ::= Num | "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" |
           "d" | "e" | "f"
419.Ostring ::= "'" {Oct} "'" "O"
420.Oct ::= Hex Hex
421.Cstring ::= """ {Char} """
422.Char ::= /* REFERENCE - A character defined by the relevant CharacterString type. For charstring
a character from the character set defined in ITU-T T.50. For universal charstring a character from
any character set defined in ISO/IEC 10646 */
423.Identifier ::= Alpha {AlphaNum | Underscore}
424.Alpha ::= UpperAlpha | LowerAlpha
425.AlphaNum ::= Alpha | Num
426.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"
427.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"
428.ExtendedAlphaNum ::= /* REFERENCE - A graphical character from the BASIC LATIN or from the
LATIN-1 SUPPLEMENT character sets defined in ISO/IEC 10646 (characters from char (0,0,0,32) to char
(0,0,0,126), from char (0,0,0,161) to char (0,0,0,172) and from char (0,0,0,174) to char (0,0,0,255)
*/
429.FreeText ::= """ {ExtendedAlphaNum} """
430.AddressValue ::= "null"
431.OmitKeyword ::= "omit"
```

A.1.6.7 Parameterization

```
432.InParKeyword ::= "in"
433.OutParKeyword ::= "out"
434.InOutParKeyword ::= "inout"
435.FormalValuePar ::= [(InParKeyword |
                        InOutParKeyword |
                        OutParKeyword
                       )] Type Identifier [":=" (Expression | Minus)]
436.FormalPortPar ::= [InOutParKeyword] Identifier Identifier

/* The first Identifier refers to the port type. The second Identifier refers to the port parameter
identifier */
437.FormalTimerPar ::= [InOutParKeyword] TimerKeyword Identifier
438.FormalTemplatePar ::= [(InParKeyword |
                           OutParKeyword |
                           InOutParKeyword
                          )] (TemplateKeyword | RestrictedTemplate) Type
                         Identifier [":=" (InLineTemplate | Minus)]
439.RestrictedTemplate ::= OmitKeyword | (TemplateKeyword TemplateRestriction)
440.TemplateRestriction ::= "(" (OmitKeyword |
                                 ValueKeyword |
                                 PresentKeyword
                                ) ")"
```

A.1.6.8 Statements

A.1.6.8.1 With statement

```
441.WithStatement ::= WithKeyword WithAttribList
442.WithKeyword ::= "with"
443.WithAttribList ::= "{" MultiWithAttrib "}"
444.MultiWithAttrib ::= {SingleWithAttrib [SemiColon]}
445.SingleWithAttrib ::= AttribKeyword [OverrideKeyword] [AttribQualifier]
                         FreeText
446.AttribKeyword ::= EncodeKeyword |
                      VariantKeyword |
                      DisplayKeyword |
                      ExtensionKeyword |
                      OptionalKeyword
447.EncodeKeyword ::= "encode"
448.VariantKeyword ::= "variant"
449.DisplayKeyword ::= "display"
450.ExtensionKeyword ::= "extension"
451.OverrideKeyword ::= "override"
452.AttribQualifier ::= "(" DefOrFieldRefList ")"
453.DefOrFieldRefList ::= DefOrFieldRef {"," DefOrFieldRef}
454.DefOrFieldRef ::= DefinitionRef |
                      (FieldReference [ExtendedFieldReference]) |
                      ("[" Minus | SingleExpression "]") |
                      AllRef
455.DefinitionRef ::= Identifier FullGroupIdentifier
456.AllRef ::= (GroupKeyword AllKeyword [ExceptKeyword "{" FullGroupIdentifierList
                                        "}"]) | ((TypeDefKeyword |
                                                  TemplateKeyword |
                                                  ConstKeyword |
                                                  AltstepKeyword |
                                                  TestcaseKeyword |
                                                  FunctionKeyword |
                                                  SignatureKeyword |
                                                  ModuleParKeyword
                                                 ) AllKeyword [ExceptKeyword
                                                               "{" IdentifierList
                                                               "}"])
```

A.1.6.8.2 Behaviour statements

```
457.BehaviourStatements ::= TestcaseInstance |
                            FunctionInstance |
                            ReturnStatement |
                            AltConstruct |
                            InterleavedConstruct |
                            LabelStatement |
                            GotoStatement |
                            RepeatStatement |
                            DeactivateStatement |
                            AltstepInstance |
                            ActivateOp |
                            BreakStatement |
                            ContinueStatement
458.SetLocalVerdict ::= SetVerdictKeyword "(" SingleExpression {"," LogItem}
                        ")"
459.SetVerdictKeyword ::= "setverdict"
460.GetLocalVerdict ::= "getverdict"
461.SUTStatements ::= ActionKeyword "(" ActionText {StringOp ActionText}
                      ")"
462.ActionKeyword ::= "action"
463.ActionText ::= FreeText | Expression
464.ReturnStatement ::= ReturnKeyword [Expression | InLineTemplate]

/* STATIC SEMANTICS - Expression shall evaluate to a value of a type compatible with the return type
for functions returning a value. It shall evaluate to a value, template (literal or template
instance), or a matching mechanism compatible with the return type for functions returning a
template. */
465.AltConstruct ::= AltKeyword "{" AltGuardList "}"
466.AltKeyword ::= "alt"
467.AltGuardList ::= {GuardStatement | ElseStatement [SemiColon]}
468.GuardStatement ::= AltGuardChar (AltstepInstance [StatementBlock] |
                                    GuardOp StatementBlock)
469.ElseStatement ::= "[" ElseKeyword "]" StatementBlock
470.AltGuardChar ::= "[" [BooleanExpression] "]"
471.GuardOp ::= TimeoutStatement |
              ReceiveStatement |
              TriggerStatement |
              GetCallStatement |
              CatchStatement |
              CheckStatement |
              GetReplyStatement |
              DoneStatement |
              KilledStatement
472.InterleavedConstruct ::= InterleavedKeyword "{" InterleavedGuardList
                             "}"
473.InterleavedKeyword ::= "interleave"
474.InterleavedGuardList ::= {InterleavedGuardElement [SemiColon]}+
475.InterleavedGuardElement ::= InterleavedGuard StatementBlock
476.InterleavedGuard ::= "[" "]" GuardOp
477.LabelStatement ::= LabelKeyword Identifier
478.LabelKeyword ::= "label"
479.GotoStatement ::= GotoKeyword Identifier
480.GotoKeyword ::= "goto"
481.RepeatStatement ::= "repeat"
482.ActivateOp ::= ActivateKeyword "(" AltstepInstance ")"
483.ActivateKeyword ::= "activate"
484.DeactivateStatement ::= DeactivateKeyword ["(" ComponentOrDefaultReference
                                               ")"]
485.DeactivateKeyword ::= "deactivate"
486.BreakStatement ::= "break"
487.ContinueStatement ::= "continue"
```

A.1.6.8.3 Basic statements

```
488.BasicStatements ::= Assignment |
                        LogStatement |
                        LoopConstruct |
                        ConditionalConstruct |
                        SelectCaseConstruct |
                        StatementBlock
489.Expression ::= SingleExpression | CompoundExpression
490.CompoundExpression ::= FieldExpressionList | ArrayExpression

/* STATIC SEMANTICS - Within CompoundExpression the ArrayExpression can be used for Arrays, record,
record of and set of types. */
491.FieldExpressionList ::= "{" FieldExpressionSpec {"," FieldExpressionSpec}
```

```
                              "}"
492.FieldExpressionSpec ::= FieldReference AssignmentChar NotUsedOrExpression
493.ArrayExpression ::= "{" [ArrayElementExpressionList] "}"
494.ArrayElementExpressionList ::= NotUsedOrExpression {"," NotUsedOrExpression}
495.NotUsedOrExpression ::= Expression | Minus
496.ConstantExpression ::= SingleExpression | CompoundConstExpression
497.BooleanExpression ::= SingleExpression

/* STATIC SEMANTICS - BooleanExpression shall resolve to a Value of type Boolean */
498.CompoundConstExpression ::= FieldConstExpressionList | ArrayConstExpression

/* STATIC SEMANTICS - Within CompoundConstExpression the ArrayConstExpression can be used for
arrays, record, record of and set of types. */
499.FieldConstExpressionList ::= "{" FieldConstExpressionSpec {"," FieldConstExpressionSpec}
                                "}"
500.FieldConstExpressionSpec ::= FieldReference AssignmentChar ConstantExpression
501.ArrayConstExpression ::= "{" [ArrayElementConstExpressionList] "}"
502.ArrayElementConstExpressionList ::= ConstantExpression {"," ConstantExpression}
503.Assignment ::= VariableRef AssignmentChar (Expression | TemplateBody)

/* STATIC SEMANTICS - The Expression on the right hand side of Assignment shall evaluate to an
explicit value of a type compatible with the type of the left hand side for value variables and
shall evaluate to an explicit value, template (literal or a template instance) or a matching
mechanism compatible with the type of the left hand side for template variables. */
504.SingleExpression ::= XorExpression {"or" XorExpression}

/* STATIC SEMANTICS - If more than one XorExpression exists, then the XorExpressions shall evaluate
to specific values of compatible types */
505.XorExpression ::= AndExpression {"xor" AndExpression}

/* STATIC SEMANTICS - If more than one AndExpression exists, then the AndExpressions shall evaluate
to specific values of compatible types */
506.AndExpression ::= NotExpression {"and" NotExpression}

/* STATIC SEMANTICS - If more than one NotExpression exists, then the NotExpressions shall evaluate
to specific values of compatible types */
507.NotExpression ::= ["not"] EqualExpression

/* STATIC SEMANTICS - Operands of the not operator shall be of type boolean or derivatives of type
Boolean. */
508.EqualExpression ::= RelExpression {EqualOp RelExpression}

/* STATIC SEMANTICS - If more than one RelExpression exists, then the RelExpressions shall evaluate
to specific values of compatible types */
509.RelExpression ::= ShiftExpression [RelOp ShiftExpression]

/* STATIC SEMANTICS - If both ShiftExpressions exist, then each ShiftExpression shall evaluate to a
specific integer, Enumerated or float Value or derivatives of these types */
510.ShiftExpression ::= BitOrExpression {ShiftOp BitOrExpression}

/* STATIC SEMANTICS - Each Result shall resolve to a specific Value. If more than one Result exists
the right-hand operand shall be of type integer or derivatives and if the shift op is "<<" or ">>"
then the left-hand operand shall resolve to either bitstring, hexstring or octetstring type or
derivatives of these types. If the shift op is "
511.BitOrExpression ::= BitXorExpression {"or4b" BitXorExpression}

/* STATIC SEMANTICS - If more than one BitXorExpression exists, then the BitXorExpressions shall
evaluate to specific values of compatible types */
512.BitXorExpression ::= BitAndExpression {"xor4b" BitAndExpression}

/* STATIC SEMANTICS - If more than one BitAndExpression exists, then the BitAndExpressions shall
evaluate to specific values of compatible types */
513.BitAndExpression ::= BitNotExpression {"and4b" BitNotExpression}

/* STATIC SEMANTICS - If more than one BitNotExpression exists, then the BitNotExpressions shall
evaluate to specific values of compatible types */
514.BitNotExpression ::= ["not4b"] AddExpression

/* STATIC SEMANTICS - If the not4b operator exists, the operand shall be of type bitstring,
octetstring or hexstring or derivatives of these types. */
515.AddExpression ::= MulExpression {AddOp MulExpression}

/* STATIC SEMANTICS - Each MulExpression shall resolve to a specific Value. If more than one
MulExpression exists and the AddOp resolves to StringOp then the MulExpressions shall be valid
operands for StringOp. If more than one MulExpression exists and the AddOp does not resolve to
StringOp then the MulExpression shall both resolve to type integer or float or derivatives of these
types.*/
```

```
516.MulExpression ::= UnaryExpression {MultiplyOp UnaryExpression}

/* STATIC SEMANTICS - Each UnaryExpression shall resolve to a specific Value. If more than one
UnaryExpression exists then the UnaryExpressions shall resolve to type integer or float or
derivatives of these types. */
517.UnaryExpression ::= [UnaryOp] Primary

/* STATIC SEMANTICS - Primary shall resolve to a specific Value of type integer or float or
derivatives of these types.*/
518.Primary ::= OpCall |
                Value |
                "(" SingleExpression ")"
519.ExtendedFieldReference ::= {(Dot Identifier) |
                                ArrayOrBitRef |
                                ("[" Minus "]")
                               }+

/* STATIC SEMANTIC - The Identifier refers to a type definition if the type of the VarInstance or
ReferencedValue in which the ExtendedFieldReference is used is anytype. ArrayOrBitRef shall be used
when referencing elements of values or arrays. The square brackets with dash shall be used when
referencing inner types of a record of or set of type. */
520.OpCall ::= ConfigurationOps |
               GetLocalVerdict |
               TimerOps |
               TestcaseInstance |
               (FunctionInstance [ExtendedFieldReference]) |
               (TemplateOps [ExtendedFieldReference]) |
               ActivateOp
521.AddOp ::= "+" |
              "-" |
              StringOp

/* STATIC SEMANTICS - Operands of the "+" or "-" operators shall be of type integer or float or
derivations of integer or float (i.e. subrange) */
522.MultiplyOp ::= "*" | "/" | "mod" | "rem"

/* STATIC SEMANTICS - Operands of the "*", "/", rem or mod operators shall be of type integer or
float or derivations of integer or float (i.e. subrange) */
523.UnaryOp ::= "+" | "-"

/* STATIC SEMANTICS - Operands of the "+" or "-" operators shall be of type integer or float or
derivations of integer or float (i.e. subrange) */
524.RelOp ::= "<" | ">" | ">=" | "<="

/* STATIC SEMANTICS - the precedence of the operators is defined in Table 6 */
525.EqualOp ::= "==" | "!="
526.StringOp ::= "&"

/* STATIC SEMANTICS - Operands of the list operator shall be bitstring, hexstring, octetstring,
(universal) character string, record of, set of, or array types, or derivates of these types */
527.ShiftOp ::= "<<" | ">>" | "<@" | "@>"
528.LogStatement ::= LogKeyword "(" LogItem {"," LogItem} ")"
529.LogKeyword ::= "log"
530.LogItem ::= FreeText | InLineTemplate
531.LoopConstruct ::= ForStatement |
                     WhileStatement |
                     DoWhileStatement
532.ForStatement ::= ForKeyword "(" Initial SemiColon BooleanExpression
                    SemiColon Assignment ")" StatementBlock
533.ForKeyword ::= "for"
534.Initial ::= VarInstance | Assignment
535.WhileStatement ::= WhileKeyword "(" BooleanExpression ")" StatementBlock
536.WhileKeyword ::= "while"
537.DoWhileStatement ::= DoKeyword StatementBlock WhileKeyword "(" BooleanExpression
                        ")"
538.DoKeyword ::= "do"
539.ConditionalConstruct ::= IfKeyword "(" BooleanExpression ")" StatementBlock
                            {ElseIfClause} [ElseClause]
540.IfKeyword ::= "if"
541.ElseIfClause ::= ElseKeyword IfKeyword "(" BooleanExpression ")" StatementBlock
542.ElseKeyword ::= "else"
543.ElseClause ::= ElseKeyword StatementBlock
544.SelectCaseConstruct ::= SelectKeyword "(" SingleExpression ")" SelectCaseBody
545.SelectKeyword ::= "select"
546.SelectCaseBody ::= "{" {SelectCase}+ "}"
547.SelectCase ::= CaseKeyword ("(" InLineTemplate {"," InLineTemplate}
                              ")" | ElseKeyword) StatementBlock
548.CaseKeyword ::= "case"
```

```
549.ExtendedIdentifier ::= [Identifier Dot] Identifier
550.IdentifierList ::= Identifier {"," Identifier}
551.FullGroupIdentifierList ::= FullGroupIdentifier {"," FullGroupIdentifier}
```

A.1.6.9 Miscellaneous productions

```
552.Dot ::= "."
553.Minus ::= "-"
554.SemiColon ::= ";"
555.Colon ::= ":"
556.Underscore ::= "_"
557.AssignmentChar ::= ":="
```

Annex B (normative): Matching values

B.1 Template matching mechanisms

This annex specifies the matching mechanisms that may be used in TTCN-3 templates (and only in templates).

B.1.1 Matching specific values

Specific values are the basic matching mechanism of TTCN-3 templates. Specific values in templates are expressions which do not contain any matching mechanisms or wildcards.

Unless otherwise specified, a template field matches the corresponding field value if, and only if, the field value has exactly the same value as the value to which the expression in the template evaluates.

EXAMPLE:

```
// Given the message type definition
type record MyMessageType
{
    integer     field1,
    charstring  field2,
    boolean     field3 optional,
    integer     field4[4]
}

// A message template using specific values could be
template MyMessageType MyTemplate:=
{
    field1 := 3+2,          // specific value of integer type
    field2 := "My string",  // specific value of charstring type
    field3 := true,         // specific value of boolean type
    field4 := {1,2,3,4}     // specific value of integer array
}
```

B.1.2 Matching mechanisms instead of values

The following matching mechanisms may be used in place of explicit values.

B.1.2.1 Value list

Value lists specify lists of acceptable values. It can be used on values of all types. A value list may also contain templates.

A template field that uses a value list matches the corresponding field if, and only if, the field value matches any one of the values or templates in the value list. Each value or template in the value list shall be of the type declared for the template field in which this mechanism is used.

EXAMPLE:

```
template MyMessage MyTemplate:=
{
    field1 := (2,4,6),              // list of integer values
    field2 := ("String1", "String2"),   // list of charstring values
    :
    :
}
```

B.1.2.2 Complemented value list

The keyword **complement** denotes a list of values that will not be accepted as values (i.e. it is the complement of a value list). It can be used on all values of all types. A complemented value list may also contain templates.

Each value or template in the list shall be of the type declared for the template field in which the complement is used.

A template field that uses complement matches the corresponding field if and only if the field does not match any of the values or templates listed in the value list. The value list may be a single value, of course.

EXAMPLE:

```
template MyMessage MyTemplate:=
{
    complement (1,3,5), // list of unacceptable integer values
    :
    field3 not(true)        // will match false
    :
}
```

B.1.2.3 Any value

The matching symbol "?" (*AnyValue*) matches any value of the specified type. It can be used on values of all types.

A template field that uses the any value mechanism matches the corresponding field if, and only if, the field evaluates to a single element of the specified type.

EXAMPLE:

```
template MyMessage MyTemplate:=
{
    field1 := ?,        // will match any integer
    field2 := ?,        // will match any non-empty charstring value
    field3 := ?,        // will match true or false
    field4 := ?     // will match any sequence of integers
}
```

B.1.2.4 Any value or none

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

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

EXAMPLE:

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

B.1.2.5 Value range

Ranges indicate a bounded range of acceptable values, including or excluding the boundaries. When used for values of **integer** or **float** types (and integer or float subtypes), a boundary value shall be either:

a) infinity or -infinity;

b) an expression that evaluates to a specific integer or float value.

The lower boundary shall be put on the left side of the range, the upper boundary at the right side. The lower boundary shall be less than the upper boundary.

A template field that uses a range matches the corresponding field if, and only if, the field value is equal to one of the values in the range.

When used in templates or template fields of **charstring** or **universal charstring** types, the boundaries shall evaluate to valid character positions according to the coded character set table(s) of the type (e.g. the given position shall not be empty). Empty positions between the lower and the upper boundaries are not considered to be valid values of the specified range.

EXAMPLE:

```
template MyMessage MyTemplate:=
{
    field1 := (1 .. !6),    // range of integer type from 1 to 5
    :
    :
    :
}
// other entries for field1 might be (-infinity to 8) or (!12 to infinity)
```

B.1.2.6 SuperSet

SuperSet is an operation for matching that shall be used only on values of **set of** types. SuperSet is denoted by the keyword **superset**. SuperSet matches a set of values if, and only if, the set of values contains at least all of the elements defined within the SuperSet, and may contain more. The argument of SuperSet shall be of the type replicated by the set of. This argument may contain templates (including template variables) and matching mechanisms, with the exception of omit, AnyValueOrNone, superset, subset and the matching attributes (length restriction and ifpresent). However, the length matching attribute may be attached to the SuperSet itself, in which case the minimal length allowed by the length attribute shall not be less than the number of the elements in the SuperSet.

EXAMPLE:

```
type set of integer MySetOfType (0 .. 10);

template MySetOfType MyTemplate1 := superset (1, 2, 3);
// matches any sequence of integers which contains at least one occurrences of the numbers
// 1, 2 and 3 in any order and position

template MySetOfType MyTemplate2_AnyValue := superset (1, 2, ?);
// matches any sequence of integers which contains at least one occurrences of the numbers
// 1, 2 and at least one more valid integer value (i.e. between 0 and 10, inclusively), in any
// order and position

template MySetOfType MyTemplate3 := superset (1, 2, (3, 4));
// matches any sequence of integers which contains at least one occurrences of the numbers
// 1, 2 and a number with the value 3 or 4, in any order and position

template MySetOfType MyTemplate4 := superset (1, 2, complement(3, 4));
// any sequence of integers matches which contains at least one occurrences of the numbers
// 1, 2 and a valid integer value which is not 3 or 4, in any order and position

template MySetOfType MyTemplate6 := superset (1, 2, 3) length (7);
// matches any sequence of 7 integers which contains at least one occurrences of the numbers
// 1, 2 and 3 in any order and position

template MySetOfType MyTemplate7 := superset (1, 2, ?) length (7 .. infinity);
// matches any sequence of at least 7 integers which contains at least one occurrences of the
// numbers 1, 2 and 3 in any order and position

template MySetOfType MyTemplate8 := superset (1, 2, 3) length (2 .. 7);
// causes an error, the lower bound of the length attribute contradicts to the minimum number
// of elements imposed by the superset argument
```

B.1.2.7 SubSet

SubSet is an operation for matching that can be used only on values of **set of** types. SubSet is denoted by the keyword **subset**. SubSet matches a set of values if, and only if, the set of values contains only elements defined within the SubSet, and may contain less. The argument of SubSet shall be of the type replicated by the set of. This argument may contain templates (including template variables) and matching mechanisms, with the exception of omit, AnyValueOrNone, superset, subset and the matching attributes (length restriction and ifpresent). However, the length matching attribute may be attached to the SubSet itself, in which case the maximum length allowed by the length attribute shall not exceed the number of the elements in the SubSet.

EXAMPLE:

```
template MySetOfType MyTemplate1:= subset (1, 2, 3);
// matches any sequence of integers which contains zero or one occurrences of the numbers
// 1, 2 and 3 in any order and position

template MySetOfType MyTemplate1:= subset (1, 2, ?);
// matches any sequence of integers which contains zero or one occurrences of the numbers
// 1, 2 and a valid integer value (i.e. between 0 and 10, inclusive) in any order and position

template MySetOfType MyTemplate1:= subset (1, 2, (3, 4));
// matches any sequence of integers which contains zero or one occurrences of the numbers
// 1, 2 and one of the numbers 3 or 4, in any order and position

template MySetOfType MyTemplate1:= subset (1, 2, complement (3, 4));
// matches any sequence of integers which contains zero or one occurrences of the numbers
// 1, 2 and a valid integer number which is not 3 or 4, in any order and position

template MySetOfType MyTemplate1:= subset (1, 2, 3) length (2);
// matches any sequence of two integers which contains zero or one occurrences of
// the numbers 1, 2 and 3, in any order and position

template MySetOfType MyTemplate1:= subset (1, 2, ?) length (0 .. 2);
// matches any sequence of zero, one or two integers which contains zero or one occurrences of
// the numbers 1, 2 and of a valid integer value, in any order and position

template MySetOfType MyTemplate1:= subset (1, 2, 3) length (0 .. 4);
// causes an error, the upper bound of length attribute contradicts to the maximum number of
// elements imposed by the subset argument
```

B.1.2.8 Omitting optional fields

The keyword **omit** denotes that an optional field shall be absent. It can be assigned to templates, but shall only be used in fields of record and set types provided that the fields are optional.

EXAMPLE:

```
template MyMessage MyTemplate:=
{   :
    :
    field3 := omit,     // omit the optional field field3
    :
}
```

B.1.3 Matching mechanisms inside values

The following matching mechanisms may be used inside explicit values of strings, records, records of, sets, sets of and arrays.

B.1.3.1 Any element

The matching symbol "?" (*AnyElement*) is used to indicate that it replaces single elements of a string (except character strings, see table 4 for the lengths of the units being matched by "?" in a string), a **record of**, a **set of** or an array. It shall be used only within values of string types, **record of** types, **set of** types and arrays.

EXAMPLE:

```
template MyMessage MyTemplate:=
{   :
    field2 := "abcxyz",
    field3 := '10???'B,     // where each "?" may either be 0 or 1
    field4 := {1, ?, 3}     // where ? may be any integer value
}
```

NOTE: The "?" in `field4` can be interpreted as *AnyValue* as an integer value, or *AnyElement* inside a **record of**, **set of** or array. Since both interpretations lead to the same match no problem arises.

B.1.3.1.1 Using single character wildcards

If it is required to express the "?" wildcard in character strings it shall be done using character patterns (see clause B.1.5). For example: "abcdxyz", "abccxyz", "abcxxyz" etc. will all match **pattern** "abc?xyz". However, "abcxyz", "abcdefxyz", etc. will not.

B.1.3.2 Any number of elements or no element

The matching symbol "*" (*AnyElementsOrNone*) is used to indicate that it replaces none or any number of consecutive elements of a string (except character strings), a **record of**, a **set of** or an array. The "*" symbol matches the longest sequence of elements possible, according to the pattern as specified by the symbols surrounding the "*".

EXAMPLE:

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

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

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

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

B.1.3.2.1 Using multiple character wildcards

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

B.1.3.3 Permutation

Permutation is an operation for matching that shall be used only on values of **record of** types. Permutation is denoted by the keyword **permutation**. Expressions, templates and *AnyElement* and *AnyElementsOrNone* are allowed as permutation elements. Each element listed in the permutation shall be of the type replicated by the **record of** type.

A permutation without *AnyElementsOrNone* in place of a single record of element means that any series of elements is acceptable provided that there is a one to one mapping between elements in the record of and in the permutation list such that each element matches its corresponding element in the permutation list.

AnyElementsOrNone used inside permutation (directly or via reference) replaces none or any number of elements within the segment of the record of value matched by permutation. The permutation matching is successful, if a subset of the elements in the record of matches the permutation list without the *AnyElementsOrNone*. If both permutation and *AnyElementsOrNone* are used in a record of template, they shall be evaluated jointly.

NOTE 1: *AnyElementsOrNone* used inside permutation has a different effect as *AnyElementsOrNone* used in conjunction with permutation as in the latter *AnyElementsOrNone* replaces consecutive elements only. For example, {**permutation**(1,2,*)} is equivalent to ({*,1,*,2,*},{*,2,*,1,*}), while {**permutation**(1,2),*} is equivalent to ({1,2},{2,1},*).

NOTE 2: When *AnyElementsOrNone* is inside a permutation, a length attribute may be applied to *AnyElementsOrNone* to restrict the number of elements matched by *AnyElementsOrNone* (see also clause B.1.4.1).

EXAMPLE:

```
type record of integer MySequenceOfType;

template MySequenceOfType MyTemplate1 := { permutation ( 1, 2, 3 ), 5 };
// matches any of the following sequences of 4 integers: 1,2,3,5; 1,3,2,5; 2,1,3,5;
// 2,3,1,5; 3,1,2,5; or 3,2,1,5

template MySequenceOfType MyTemplate2 := { permutation ( 1, 2, ? ), 5 };
// matches any sequence of 4 integers that ends with 5 and contains 1 and 2 at least once in
// other positions

template MySequenceOfType MyTemplate3 := { permutation ( 1, 2, 3 ), * };
// matches any sequence of integers starting with 1,2,3; 1,3,2; 2,1,3; 2,3,1; 3,1,2 or 3,2,1

template MySequenceOfType MyTemplate4 := { *, permutation ( 1, 2, 3 )};
// matches any sequence of integers ending with 1,2,3; 1,3,2; 2,1,3; 2,3,1; 3,1,2 or 3,2,1

template MySequenceOfType MyTemplate5 := { *, permutation ( 1, 2, 3 ),* };
// matches any sequence of integers containing any of the following substrings at any position:
// 1,2,3; 1,3,2; 2,1,3; 2,3,1; 3,1,2 or 3,2,1

template MySequenceOfType MyTemplate6 := { permutation ( 1, 2, * ), 5 };
// matches any sequence of integers that ends with 5 and containing 1 and 2 at least once in
// other positions

template MySequenceOfType MyTemplate7 := { permutation ( 1, 2, 3 ), * length (0..5)};
// matches any sequence of three to eight integers starting with 1,2,3; 1,3,2; 2,1,3; 2,3,1;
// 3,1,2 or 3,2,1

template integer MyInt1 := (1,2,3);
template integer MyInt2 := (1,2,?);
template integer MyInt3 := ?;
template integer MyInt4 := *;

template MySequenceOfType MyTemplate10 := { permutation (MyInt1, 2, 3 ), 5 };
// matches any of the sequences of 4 integers:
//      1,3,2,5; 2,1,3,5; 2,3,1,5; 3,1,2,5; or 3,2,1,5;
//      2,3,2,5; 2,2,3,5; 2,3,2,5; 3,2,2,5; or 3,2,2,5;
//      3,3,2,5; 2,3,3,5; 2,3,3,5; 3,3,2,5; or 3,2,3,5;

template MySequenceOfType MyTemplate11 := { permutation (MyInt2, 2, 3 ), 5 };
// matches any sequence of 4 integers that ends with 5 and contains 2 and 3 at least once in
// other positions

template MySequenceOfType MyTemplate12 := { permutation (MyInt3, 2, 3 ), 5 };
// matches any sequence of 4 integers that ends with 5 and contains 2 and 3 at least once in
// other positions

template MySequenceOfType MyTemplate13 := { permutation (MyInt4, 2, 3 ), 5 };
// matches any sequence of integers that ends with 5 and containing 2 and 3 at least once in
// other positions

template MySequenceOfType MyTemplate14 := { permutation (MyInt3, 2, ? ), 5 };
// matches any sequence of 4 integers that ends with 5 and contains 2 at least once in
// other positions

template MySequenceOfType MyTemplate15 := { permutation (MyInt4, 2, * ), 5 };
// matches any sequence of integers that ends with 5 and contains 2 at least once in
// other positions
```

B.1.4 Matching attributes of values

The following attributes may be associated with matching mechanisms.

B.1.4.1 Length restrictions

The **length** restriction attribute is used to restrict the length of string values matching the template or the number of elements in a **set of**, **record of** or array structure. It shall be used only as an attribute of the following matching mechanisms: ValueOrAttribList, ComplementedList, *AnyValue*, *AnyValueOrNone*, *AnyElement*, *AnyElementsOrNone*, superset, subset, and pattern. It can also be used in conjunction with the **ifpresent** matching attribute. The syntax for **length** can be found in clauses 6.2.3 and 6.3.3.

NOTE: When the **length** attribute is used with a value list, elements of the list may be disabled by the attribute.

When both the complement and the length restriction matching mechanisms are used for a template or template field, restrictions implied by them shall apply to the template or template field independently.

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

The length specifications for the template shall not conflict with the length for restrictions (if any) of the corresponding type.

A template field that uses length as an attribute of a symbol matches the corresponding field if, and only if, the field matches both the symbol and its associated attribute. The length attribute matches if the length of the field is greater than or equal to the specified lower bound and less than or equal to the upper bound. In the case of a single length value the length attribute matches only if the length of the received field is exactly the specified value.

It is allowed to use a length restriction in conjunction with the special value **omit**, however in this case the length attribute has no effect (i.e. with **omit** it is redundant). With *AnyValueOrNone* and **ifpresent** it places a restriction on the value, if any.

EXAMPLE:

```
template Mymessage MyTemplate:=
{
    field1 := complement ({4,5},{1,4,8,9}) length (1 .. 6), // any value containing 1, 2, 3, 4,
    // 5 or 6 elements is accepted provided it is not {4,5} or {1,4,8,9}
    field2 := "ab*ab" length(5),   // matches the character string "ab*ab" only
    field3 := "ab*ab" length(13),  // never matches as the specific value is of length 5
                                   // and not of length 13
    field4 := pattern "ab*ab" length(13),
                                // max length of the AnyElementsOrNone string is 9 characters
    :
}
```

B.1.4.2 The IfPresent indicator

The **ifpresent** indicates that a match may be made if an optional field is present (i.e. not omitted). This attribute may be used with all the matching mechanisms, provided this field is declared as optional.

A template field that uses **ifpresent** matches the corresponding field if, and only if, the field matches according to the associated matching mechanism, or if the field is absent.

EXAMPLE:

```
template Mymessage MyTemplate:=
{   :
    field2 := "abcd" ifpresent, // matches "abcd" if not omitted
    :
    :
}
```

NOTE: *AnyValueOrNone* has exactly the same meaning as **? ifpresent**.

B.1.5 Matching character pattern

Character patterns can be used in templates to define the format of a required character string to be received. Character patterns can be used to match **charstring** and **universal charstring** values. In addition to literal characters, character patterns allow the use of meta-characters (e.g. ? and * within a character pattern means matching any character and any number of any character respectively).

EXAMPLE 1:

```
template charstring MyTemplate:= pattern "ab??xyz*0";
```

This template would match any character string that consists of the characters "ab", followed by any two characters, followed by the characters "xyz", followed by any number of any characters (including any number of "0"-s) before the closing character "0".

If it is required to interpret any metacharacter literally it shall be preceded with the metacharacter "\".

EXAMPLE 2:

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

This template would match any character string which consists of the characters "ab", followed by any character, followed by the characters "?xyz", followed by any number of any characters.

The list of meta characters for TTCN-3 patterns is shown in table B.1. Metacharacters shall not contain whitespaces except a whitespace preceded by a newline character before or inside a set expression.

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

Metacharacter	Description
?	Match any character (see notes 1 and 2)
*	Match any character zero or more times; shall match the longest possible number of characters (see example 1 above) (see notes 1 and 2)
\	Cause the following metacharacter to be interpreted as a literal (see note 3). When preceding a character without defined metacharacter meaning "\" and the character together match the character following the "\" (see note 4)
[ ]	Match any character within the specified set, see clause B.1.5.1 for more details
-	Has a metacharacter meaning inside a pair of square brackets ("[" and "]") only, except the first and last positions within the bracket. Allows to specify a range of characters; see clause B.1.5.1 for more details
^	Has a metacharacter meaning as the first character following the opening square bracket inside a pair of square brackets ("[" and "]") only and cause to match any character complementing the set of characters following this metacharacter; see clause B.1.5.1 for more details
\q{group,plane,row,cell}	Match the Universal character specified by the quadruple
{reference}	Insert the referenced user defined string and interpret it as a regular expression. See clause B.1.5.2 for more details
\N{reference}	Match any character within the set of characters, where the set is defined by the referenced definition; see clause B.1.5.4 for more details
\d	Match any numerical digit (equivalent to [0-9])
\w	Match any alphanumeric character (equivalent to [0-9a-zA-Z])
\t	Match the C0 control character HT(9) (see ITU-T Recommendation T.50 [4])
\n	Match any of the following C0 control characters: LF(10), VT(11), FF(12), CR(13) (see ITU-T Recommendation T.50 [4]) (jointly called newline characters, see clause A.1.5.1)
\r	Match the C0 control character CR (see ITU-T Recommendation T.50 [4])
\s	Match any one of the following C0 control characters: HT(9), LF(10), VT(11), FF(12), CR(13), SP(32) (see ITU-T Recommendation T.50 [4]) (jointly called white-space characters, see clause A.1.5.1)
\b	Match a word boundary (any graphical character except SP or DEL is preceded or followed by any of the whitespace or newline characters)
\"	Match the double quote character
""	Match the double quote character
|	Used to denote two alternative expressions
( )	Used to group an expression
#(n, m)	Match the preceding expression at least n times but no more than m times (postfix). See clause B.1.5.3 for more details
#n	Match the previous expression exactly n times (where n is a single digit) (postfix); the same as #(n)
+	Match the preceding expression one or several times (postfix); the same as #(1,)

Metacharacter	Description
NOTE 1:	Metacharacters ? and * are able to match any characters of the character set of the root type of the template or template field in which they are used (i.e. not considering type constraints applied). However, it shall not be forgotten, that receiving operations require type checking of the received message before attempting to match it. Therefore received values not complying with the subtype specification of the template or template field are never provided for matching.
NOTE 2:	In some other languages/notations ? and * has different meaning as metacharacters. However in TTCN these characters are traditionally used for matching in the sense as specified in this table.
NOTE 3:	Consequently the backslash character can be matched by a pair of backslash characters without space between them (\\), e.g. the pattern "\\d" will match the string "\d"; opening or closing square brackets can be matched by "\[" and "\]" respectively, etc.
NOTE 4:	Such use of the metacharacter "\" is deprecated as further metacharacters can be defined later.

Character patterns may be composed from several fragments using the concatenation operation. The fragments of the pattern shall be concatenated before any evaluation of the pattern expression. See also the shorthand notation for referenced definitions at concatenation in clause B.1.5.2.

EXAMPLE 3:

```
template charstring MyTemplate:= pattern "ab?\?" & "xyz*"; // results in the same pattern as
                                                           // in example2
```

B.1.5.1 Set expression

A list of characters enclosed by a pair of "[" and "]" matches any single character in that list. The set expression is delimited by the "[" "]" symbols. In addition to character literals, it is possible to specify character ranges using the hyphen "-" as separator. The range consist of the character immediately before the separator, the character immediately after it and all characters with a character code between the codes of the two bordering characters. A hyphen character "-" inside the list but without preceding or following character loses its special meaning.

The set expression can also be negated by placing the caret "^" character as the first character after the opening square bracket. Negation takes precedence over character ranges. Therefore a hyphen "-" immediately following a negating caret "^" shall be processed as a literal character.

An empty list and an empty negated list are not allowed. Therefore a closing square bracket "]" immediately following an opening square bracket "[" or a caret following the opening square bracket "[" and immediately followed by a closing square bracket "]" shall be processed as literal characters.

All metacharacters, except those listed below, lose their special meaning inside the list:

- "]" not at the first position and not immediately following a "^" at the first position;
- "-" not at the first or last positions in the list;
- "^" at the first position in the list except when immediately followed by a closing square bracket;
- "\", "\d", "\t", "\w", "\r", "\n", "\s" and "\b";
- "\q{group,plane,row,cell}";
- "**N**{reference}".

NOTE 1: Embedded lists are not allowed (for example in pattern "[ab[r-z]]" the second "[" denotes a literal "[", the first "]" closes the list and the second "]" causes an error as no related opening bracket in the pattern).

NOTE 2: To include a literal caret character "^", place it anywhere except in the first position or precede it with a backslash. To include a literal hyphen "-", place it first or last in the list, or precede it with a backslash. To include a literal closing square bracket "]", place it first or precede it with a backslash. If the first character in the list is the caret "^", then the characters "-" and "]" also match themselves when they immediately follow that caret.

EXAMPLE:

```
template charstring RegExp1:= pattern "[a-z]";  // this will match any character from a to z

template charstring RegExp2:= pattern "[^a-z]";  // this will match any character except a to z

template charstring RegExp3:= pattern "[AC-E][0-9][0-9][0-9]YKE";

// RegExp3 will match a string which starts with the letter A or a letter between
// C and E (but not e.g. B) then has three  digits and the letters YKE
```

B.1.5.2 Reference expression

In addition to direct string values, it is also possible within the pattern to use references to templates, constants, variables, formal parameters, module parameters, or to their fields. The reference shall be enclosed within the "{" "}" characters and reference shall resolve a compatible character string type. Contents of the referenced templates, constants or variables shall be handled as a regular expression. Each expression shall be dereferenced only once, before the insertion (i.e. the expression dereferenced and inserted into the referencing pattern shall not be dereferenced again).

EXAMPLE 1:

```
const charstring MyString:= "ab?";

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

This template would match any character string that consists of the characters "ab" followed by any character.

```
template universal charstring MyTemplate1:= pattern "{MyString}de\q{1, 1, 13, 7}";
```

This template would match any character string which consists of the characters "ab", followed by any character, followed by the characters "de", followed by the character in ISO10646-1 with group=1, plane=1, row=13 and cell=7.

If a referenced definition or field of a definition contains one or more reference expressions, then these references shall recursively be dereferenced before inserting their contents into the referencing pattern.

If a fragment of a pattern contains a single reference only, it is allowed, as a shorthand notation, to reference the definition or the field of the definition directly, i.e. leave out double quotes (" ") and the pair of curly brackets ({ }).

EXAMPLE 2:

```
const charstring MyConst2 := "ab";
template charstring RegExp1 := pattern "{MyConst2}";
   // matches the string "ab"
template charstring RegExp1a := pattern MyConst2;
   // the same as above, matches the string "ab"
template charstring RegExp2 := pattern "{RegExp1}{RegExp1}";
   // matches the string "abab"
template charstring RegExp2a := pattern "{RegExp1}" & "{RegExp1}";
   // the same as above, matches the string "abab"
template charstring RegExp2b := pattern RegExp1 & RegExp1;
   // the same as above, matches the string "abab"
template charstring RegExp3 := pattern "c{RegExp2}d";
   // matches the string "cababd"

template charstring RegExp4 := pattern "{Reg";
template charstring RegExp5 := pattern "Exp1}";
template charstring RegExp6 := pattern "{RegExp4}{RegExp5}";
   // matches the string "{RegExp1}" only (i.e. shall not be handled as a reference expression
   // after insertion)
template charstring RegExp7 := pattern "{Reg" & "Exp1}";
   // note the difference to the previous example; in this case the fragments of the
   // pattern are joined before any evaluation, i.e. this template will match the string "ab"
```

EXAMPLE 3:

```
template charstring Ref0:= "My String";
template charstring Ref1:= "{Re";
template charstring Ref2:= "f0}";
template charstring Ref3:= "{Ref1}{Ref2}";
   //this matches "{Ref0}"
   //i.e. there is no further dereferencing
   //as Ref1 and Ref2 do not contain a reference
```

```
template charstring Ref4:= "{Ref0}";
template charstring Ref5:= "";
template charstring Ref6:= "{Ref4}{Ref5}";
   //this matches "My String" – here Ref0 is dereferenced, because Ref4 contains
   //the reference expression {Ref0} with the reference Ref0
```

EXAMPLE 4:

```
type record MyRecord {
    integer i,
    charstring c
}
const MyRecord referencedRecord:= {1,"this"}
const charstring referencedConstant := referencedRecord.c;
template charstring referencingPattern := pattern "{referencedConstant}"
   //this matches "this" as the referencedConstant is dereferenced
```

B.1.5.3 Match expression n times

To specify that the preceding expression should be matched a number of times one of the following syntaxes shall be used: "#(n, m)", "#(n,)", "#(, m)", "#(n)", "#n" or "+".. The form "#(n, m)" specifies that the preceding expression must be matched at least n times but not more than m times. The metacharacter postfix "#(n,)" specifies that the preceding expression must be matched at least n times while "#(, m)" indicates that the preceding expression shall be matched not more than m times. Metacharacters (postfixes) "#(n)" and "#n" specify that the preceding expression must be matched exactly n times (they are equivalent to "#(n, n)"). In the form "#n" n shall be a single digit. The metacharacter postfix "+" denotes that the preceding expression must be matched at least 1 time (equivalent to "#(1,)").

EXAMPLE:

```
template charstring RegExp4:= pattern "[a-z]#(9, 11)"; // match at least 9 but no more than 11
                                                       // characters from a to z
template charstring RegExp5a:= pattern "[a-z]#(9)";     // match exactly 9
                                                       // characters from a to z
template charstring RegExp5b:= pattern "[a-z]#9";   // match exactly 9
                                                       // characters from a to z
template charstring RegExp6:= pattern "[a-z]#(9, )";  // match at least 9
                                                       // characters from a to z
template charstring RegExp7:= pattern "[a-z]#(, 11)"; // match no more than 11
                                                       // characters from a to z
template charstring RegExp8:= pattern "[a-z]+";     // match at least 1
                                                       // characters from a to z,
```

B.1.5.4 Match a referenced character set

A notation of the form **"\N{**reference**}"**, where reference is denoting a one-character-length template, constant, variable, formal parameter or module parameter, matches the character in the referenced value or template.

Referencing a template, constant, variable, formal parameter or module parameter that is not of length 1 shall cause an error.

A notation of the form **"\N{**typereference**}"**, where "typereference" is a reference to a **charstring** or **universal charstring** type, matches any character of the character set denoted by the referenced type.

NOTE 1: Cases when the referenced set of characters is not a subset of values allowed by the type definition of the template or template field for which the character pattern is used, are not be treated as an error (but e.g. matching never can occur if the two sets do not overlap).

NOTE 2: \N{**charstring**} is equivalent to ? when the latter is applied to a template or template field of **charstring** type and \N{**universal charstring**} is equivalent to ? when the latter is applied to a template or template field of **universal charstring** type (but causes an error if applied to a template or template field of **charstring** type).

EXAMPLE:

```
type charstring MyCharRange ("a".."z");
type charstring MyCharList ("a", "z");
const MyCharRange myCharR := "r";

template charstring myTempPatt1 := pattern "\N{myCharR}";
// myTempPatt1 shall match the string "r" only

template charstring myTempPatt2 := pattern "\N{MyCharRange}";
// myTempPatt2 shall match any string containing a single character from a to z

template MyCharRange myTempPatt3 := pattern "\N{MyCharList}";
// myTempPatt3 shall match strings "a" or "z" only
```

B.1.5.5 Type compatibility rules for patterns

For the purpose of referenced patterns (see clause B.1.5.2) and references character sets (see clause B.1.5.3) specific type compatibility rules apply: a referenced type, template, constant, variable or module parameter of the type **charstring** always can be used in the pattern specification of a template or template field of **universal charstring** type; a referenced type, template or value of the type **universal charstring** can be used in the pattern specification of a template or template field of **charstring** type if all characters used in the referenced template or value and the character set allowed by the referenced type has their corresponding characters in the **charstring** type (see definition of corresponding characters in clause 6.3.1).

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

This annex defines the TTCN-3 predefined functions.

C.0 General exception handling procedures

When the general restrictions specified in clause 16.1.2 are not met, this shall cause a compile time or runtime error. Error situations for which no explicit exception-handling rule is defined in the relevant clauses of this annex shall cause a TTCN-3 compile-time or run-time error. Which error situation causes compile-time and which one run-time error is a tool implementation option.

C.1 Integer to character

```
int2char(in integer invalue) return charstring
```

This function converts an **integer** value in the range of 0 to 127 (8-bit encoding) into a single-character-length **charstring** value. The integer value describes the 8-bit encoding of the character.

In addition to the general error causes in clause 16.1.2, error causes are:

- `invalue` is less than 0 or greater than 127.

C.2 Integer to universal character

```
int2unichar(in integer invalue) return universal charstring
```

This function converts an **integer** value in the range of 0 to 2 147 483 647 (32-bit encoding) into a single-character-length **universal charstring** value. The integer value describes the 32-bit encoding of the character.

In addition to the general error causes in clause 16.1.2, error causes are:

- `invalue` is less than 0 or greater than 2147483647.

C.3 Integer to bitstring

```
int2bit(in integer invalue, in integer length) return bitstring
```

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

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

In addition to the general error causes in clause 16.1.2, error causes are:

- `invalue` is less than zero;
- the conversion yields a return value with more bits than specified by `length`.

C.4 Integer to hexstring

```
int2hex(in integer invalue, in integer length) return hexstring
```

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

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

In addition to the general error causes in clause 16.1.2, error causes are:

- `invalue` is less than zero;
- the conversion yields a return value with more hexadecimal characters than specified by `length`.

C.5 Integer to octetstring

```
int2oct(in integer invalue, in integer length) return octetstring
```

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

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

In addition to the general error causes in clause 16.1.2, error causes are:

- `invalue` is less than zero;
- the conversion yields a return value with more octets than specified by `length`.

C.6 Integer to charstring

```
int2str(in integer invalue) return charstring
```

This function converts the integer value into its string equivalent (the base of the return string is always decimal).

The general error causes in clause 16.1.2 apply.

EXAMPLE:

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

C.7 Integer to float

```
int2float(in integer invalue) return float
```

This function converts an **integer** value into a **float** value.

The general error causes in clause 16.1.2 apply.

EXAMPLE:

```
int2float(4) = 4.0
```

C.8 Float to integer

```
float2int(in float invalue) return integer
```

This function converts a **float** value into an **integer** value by removing the fractional part of the argument and returning the resulting **integer**.

In addition to the general error causes in clause 16.1.2, error causes are:

- `invalue` is **infinity**, **-infinity** or **not_a_number**.

EXAMPLE:

```
float2int(3.12345E2) = float2int(312.345) = 312
```

C.9 Character to integer

```
char2int(in charstring invalue) return integer
```

This function converts a single-character-length **charstring** value into an integer value in the range of 0 to 127. The integer value describes the 8-bit encoding of the character.

In addition to the general error causes in clause 16.1.2, error causes are:

- length of `invalue` does not equal 1.

C.10 Character to octetstring

```
char2oct(in charstring invalue) return octetstring
```

This function converts a **charstring** `invalue` to an **octetstring**. Each octet of the **octetstring** will contain the ITU-T Recommendation T.50 [4] codes (according to the IRV) of the appropriate characters of `invalue`.

The general error causes in clause 16.1.2 apply.

EXAMPLE:

```
char2oct ("Tinky-Winky") = '54696E6B792D57696E6B79'O
```

C.11 Universal character to integer

```
unichar2int(in universal charstring invalue) return integer
```

This function converts a single-character-length **universal charstring** value into an integer value in the range of 0 to 2 147 483 647. The integer value describes the 32-bit encoding of the character.

In addition to the general error causes in clause 16.1.2, error causes are:

- length of `invalue` does not equal 1.

C.12 Bitstring to integer

```
bit2int(in bitstring invalue) return integer
```

This function converts a single **bitstring** value to a single **integer** value.

For the purposes of this conversion, a **bitstring** shall be interpreted as a positive base 2 **integer** value. The rightmost bit is least significant, the leftmost bit is the most significant. The bits 0 and 1 represent the decimal values 0 and 1 respectively.

NOTE: On real test systems the integer interpretation of `invalue` may lead to an overflow problem that causes compile time or run-time error. However, this is out of the scope of the present document.

The general error causes in clause 16.1.2 apply.

C.13 Bitstring to hexstring

```
bit2hex(in bitstring invalue) return hexstring
```

This function converts a single **bitstring** value to a single **hexstring**. The resulting **hexstring** represents the same value as the **bitstring**.

For the purpose of this conversion, a bitstring shall be converted into a hexstring, where the bitstring is divided into groups of four bits beginning with the rightmost bit. Each group of four bits is converted into a hex digit as follows:

'0000'B → '0'H, '0001'B → '1'H, '0010'B → '2'H, '0011'B → '3'H, '0100'B → '4'H, '0101'B → '5'H, '0110'B → '6'H, '0111'B → '7'H, '1000'B → '8'H, '1001'B → '9'H, '1010'B → 'A'H, '1011'B → 'B'H, '1100'B → 'C'H, '1101'B → 'D'H, '1110'B → 'E'H, and '1111'B → 'F'H.

When the leftmost group of bits does contain less than 4 bits, this group is filled with '0'B from the left until it contains exactly 4 bits and is converted afterwards. The consecutive order of hex digits in the resulting hexstring is the same as the order of groups of 4 bits in the bitstring.

The general error causes in clause 16.1.2 apply.

EXAMPLE:

```
bit2hex ('111010111'B)= '1D7'H
```

C.14 Bitstring to octetstring

```
bit2oct(in bitstring invalue) return octetstring
```

This function converts a single **bitstring** value to a single **octetstring**. The resulting **octetstring** represents the same value as the **bitstring**.

For the conversion the following holds: bit2oct(value)=hex2oct(bit2hex(value)).

The general error causes in clause 16.1.2 apply.

EXAMPLE:

```
bit2oct('111010111'B)= '01D7'O
```

C.15 Bitstring to charstring

```
bit2str(in bitstring invalue) return charstring
```

This function converts a single **bitstring** value to a single **charstrin**g. The resulting **charstring** has the same length as the **bitstring** and contains only the characters '0' and '1'.

For the purpose of this conversion, a **bitstring** shall be converted into a **charstrin**g. Each bit of the **bitstring** is converted into a character '0' or '1' depending on the value 0 or 1 of the bit. The consecutive order of characters in the resulting **charstring** is the same as the order of bits in the **bitstrin**g.

The general error causes in clause 16.1.2 apply.

EXAMPLE:

```
bit2str('1110101'B) will return "1110101"
```

C.16 Hexstring to integer

```
hex2int(in hexstring invalue) return integer
```

This function converts a single **hexstring** value to a single **integer** value.

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

NOTE: On real test systems the integer interpretation of `invalue` may lead to an overflow problem that causes compile time or run-time error. However, this is out of the scope of the present document.

The general error causes in clause 16.1.2 apply.

C.17 Hexstring to bitstring

```
hex2bit(in hexstring invalue) return bitstring
```

This function converts a single **hexstring** value to a single **bitstring**. The resulting **bitstring** represents the same value as the **hexstring**.

For the purpose of this conversion, a **hexstring** shall be converted into a **bitstring**, where the hex digits of the **hexstring** are converted in groups of bits as follows:

'0'H → '0000'B, '1'H → '0001'B, '2'H → '0010'B, '3'H → '0011'B, '4'H → '0100'B, '5'H → '0101'B, '6'H → '0110'B, '7'H → '0111'B, '8'H → '1000'B, '9'H → '1001'B, 'A'H → '1010'B, 'B'H → '1011'B, 'C'H → '1100'B, 'D'H → '1101'B, 'E'H → '1110'B, and 'F'H → '1111'B.

The consecutive order of the groups of 4 bits in the resulting **bitstring** is the same as the order of hex digits in the **hexstring**.

The general error causes in clause 16.1.2 apply.

EXAMPLE:

```
hex2bit('1D7'H)= '000111010111'B
```

C.18 Hexstring to octetstring

```
hex2oct(in hexstring invalue) return octetstring
```

This function converts a single **hexstring** value to a single **octetstring**. The resulting **octetstring** represents the same value as the **hexstring**.

For the purpose of this conversion, a **hexstring** shall be converted into a **octetstring**, where the **octetstring** contains the same sequence of hex digits as the **hexstring** when the length of the **hexstring** modulo 2 is 0. Otherwise, the resulting **octetstring** contains 0 as leftmost hex digit followed by the same sequence of hex digits as in the **hexstring**.

The general error causes in clause 16.1.2 apply.

EXAMPLE:

```
hex2oct('1D7'H)= '01D7'O
```

C.19 Hexstring to charstring

```
hex2str(in hexstring invalue) return charstring
```

This function converts a single hexstring value to a single charstring. The resulting charstring has the same length as the hexstring and contains only the characters '0' to '9'and 'A' to 'F'.

For the purpose of this conversion, a **hexstring** shall be converted into a **charstring**. Each hex digit of the **hexstring** is converted into a character '0' to '9' and 'A' to 'F' depending on the value 0 to 9 or A to F of the hex digit. The consecutive order of characters in the resulting **charstring** is the same as the order of digits in the **hexstring**.

The general error causes in clause 16.1.2 apply.

EXAMPLE:

```
hex2str('AB801'H) will return "AB801"
```

C.20 Octetstring to integer

```
oct2int(in octetstring invalue) return integer
```

This function converts a single **octetstring** value to a single **integer** value.

For the purposes of this conversion, an **octetstring** shall be interpreted as a positive base 16 **integer** value. The rightmost hexadecimal digit is least significant, the leftmost hexadecimal digit is the most significant. The number of hexadecimal digits provided shall be multiples of 2 since one octet is composed of two hexadecimal digits. The hexadecimal digits 0 to F represent the decimal values 0 to 15 respectively.

NOTE: On real test systems the integer interpretation of `invalue` may lead to an overflow problem that causes compile time or run-time error. However, this is out of the scope of the present document.

The general error causes in clause 16.1.2 apply.

C.21 Octetstring to bitstring

```
oct2bit(in octetstring invalue) return bitstring
```

This function converts a single **octetstring** value to a single **bitstring**. The resulting **bitstring** represents the same value as the **octetstring**.

For the conversion the following holds: oct2bit(value)=hex2bit(oct2hex(value)).

The general error causes in clause 16.1.2 apply.

EXAMPLE:

```
oct2bit ('01D7'O)='0000000111010111'B
```

C.22 Octetstring to hexstring

```
oct2hex(in octetstring invalue) return hexstring
```

This function converts a single **octetstring** value to a single **hexstring**. The resulting **hexstring** represents the same value as the **octetstring**.

For the purpose of this conversion, a **octetstring** shall be converted into a **hexstring** containing the same sequence of hex digits as the **octetstring**.

The general error causes in clause 16.1.2 apply.

EXAMPLE:

```
oct2hex('1D74'O)= '1D74'H
```

C.23 Octetstring to character string

```
oct2str(in octetstring invalue) return charstring
```

This function converts an **octetstring** `invalue` to an **charstring** representing the string equivalent of the input value. The resulting **charstring** shall have the same length as the incoming **octetstring**.

For the purpose of this conversion each hex digit of `invalue` is converted into a character '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E' or 'F' echoing the value of the hex digit. The consecutive order of characters in the resulting **charstring** is the same as the order of hex digits in the **octetstring**.

The general error causes in clause 16.1.2 apply.

EXAMPLE:

```
oct2str('4469707379'O) = "4469707379"
```

C.24 Octetstring to character string, version II

```
oct2char(in octetstring invalue) return charstring
```

This function converts an **octetstring** `invalue` to a **charstring**. The input parameter `invalue` shall not contain octet values higher than 7F. The resulting **charstring** shall have the same length as the input **octetstring**. The octets are interpreted as ITU-T Recommendation T.50 [4] codes (according to the IRV) and the resulting characters are appended to the returned value.

The general error causes in clause 16.1.2 apply.

EXAMPLE:

```
oct2char('4469707379'O) = "Dipsy"
```

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

C.25 Charstring to integer

```
str2int(in charstring invalue) return integer
```

This function converts a **charstring** representing an **integer** value to the equivalent **integer**.

In addition to the general error causes in clause 16.1.2, error causes are:

- invalue contains characters other than "0", "1", "2", "3", "4", "5", "6", "7", "8", "9" and "-".
- invalue contains the character "-" at another position than the leftmost one.

NOTE: On real test systems the integer interpretation of invalue may lead to an overflow problem that causes compile time or run-time error. However, this is out of the scope of the present document.

EXAMPLE:

```
str2int("66")   // will return the integer value 66
str2int("-66")  // will return the integer value -66
str2int("6-6")  // will cause an error
str2int("abc")  // will cause an error
str2int("0")    // will return the integer value 0
```

C.26 Character string to hexstring

```
str2hex(in charstring invalue) return hexstring
```

This function converts a string of the type **charstring** to a **hexstring**. The string invalue shall contain the "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "a", "b", "c", "d", "e" "f", "A", "B", "C", "D", "E" or "F" graphical characters only. Each character of invalue shall be converted to the corresponding hexadecimal digit. The resulting **hexstring** will have the same length as the incoming **charstring**.

In addition to the general error causes in clause 16.1.2, error cause is:

- invalue contains characters other than specified above.

EXAMPLE:

```
str2hex("54696E6B792D57696E6B7") = '54696E6B792D57696E6B7'H
```

C.27 Character string to octetstring

```
str2oct(in charstring invalue) return octetstring
```

This function converts a string of the type **charstring** to an **octetstring**. The string invalue shall contain the "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "a", "b", "c", "d", "e" "f", "A", "B", "C", "D", "E" or "F" graphical characters only. When the string invalue contains even number characters the resulting **octetstring** contains 0 as leftmost character followed by the same sequence of characters as in the **charstring**.

lengthof (see clause C.) for the resulting **octetstring** will return half of **lengthof** of the incoming **charstring**. In addition to the general error causes in clause 16.1.2, error causes is:

- invalue contains characters other than specified above.

EXAMPLE:

```
str2oct("54696E6B792D57696E6B79") = '54696E6B792D57696E6B79'O
str2oct("1D7")= '01D7'O
```

NOTE: The semantic of the str2oct function cause asymmetric behaviour:

```
oct2str(str2oct("1D7"))// results the charstring value "01D7"
```

C.28 Character string to float

```
str2float(in charstring invalue) return float
```

This function converts a **charstring** comprising a number into a **float** value. The format of the number in the **charstring** shall follow rules in clause 6.1.0, items a) or b) with the following exceptions:

- leading zeros are allowed;
- leading "+" sign before positive values is allowed;
- "-0.0" is allowed;
- no numbers after the dot in the decimal notation are allowed.

In addition to the general error causes in clause 16.1.2, error causes are:

- the format of invalue is different than defined above.

NOTE: On real test systems the float interpretation of `invalue` may lead to an overflow problem that causes compile time or run-time error. However, this is out of the scope of the present document.

EXAMPLE:

```
str2float("12345.6")    // is the same as str2float("123.456E+02")
str2float("1.6")        // returns a float value equal to 1.6
str2float("+001.")      // returns a float value equal to 1.0
str2float("+001")       // returns a float value equal to 1.0
str2float("-0.0")       // returns a float value equal to -0.0
```

C.29 Length of strings and lists

```
lengthof(in template (present) any_string_or_list_type inpar) return integer
```

This function returns the length of a value or template that is of type **bitstring**, **hexstring**, **octetstring**, **charstring, universal charstring, record of**, **set of**, or array (see the note below). The units of length for each string type are defined in table 4 in the main body of the present document. For **record of**, **set of**, and array, the value to be returned is the sequential number of the last initialized element: in case of **record of** and **set of** the index of that element plus 1. In case of arrays, **lengthof** should return the index of that last element minus the index of the first element plus 1.

The length of a fixed length **record of**, **set of**, or array value will always be the fixed length according to the type definition.

The length of an **universal charstring** shall be calculated by counting each combining character and hangul syllable character (including fillers) on its own (see ISO/IEC 10646 [2], clauses 23 and 24).

When the function **lengthof** is applied to string-type templates, `inpar` shall only contain the following matching mechanisms: specific value, value list, complemented value list, pattern, "?" (*AnyValue* instead of value), "*" (*AnyValueOrNone* instead of value), "?" (*AnyElement* inside value) and "*" (*AnyElementsOrNone* inside value) and the length restriction matching attribute. In case of string-type templates `inpar` shall match values of the same length only.

When the function **lengthof** is applied to templates of record of or set of types, `inpar` shall only contain the following matching mechanisms: specific value, value list, complemented value list, "?" (*AnyValue* instead of value), "*" (*AnyValueOrNone* instead of value), SuperSet, SubSet, "?" (*AnyElement* inside value) and "*" (*AnyElementsOrNone* inside value), permutation and the length restriction matching attribute. The parameter `inpar` shall only match values, for which the **lengthof** function would give the same result.

NOTE 1: In case of record ofs and set ofs and arrays only elements of the TTCN-3 object, which is the parameter of the function are calculated; i.e. no elements of nested types are taken into account at determining the return value.

In addition to the general error causes in clause 16.1.2, error causes are:

- `inpar` is a string-type template and it can match string values with different length or the length restriction matching attribute contradicts the number of string elements in the template body;
- `inpar` is a record of or set of type template and it can match values of different lengths or the length restriction matching attribute contradicts the number of elements in the template body.

NOTE 2: On real test systems the length calculation of `inpar` may lead to an overflow problem that causes compile time or run-time error. However, this is out of the scope of the present document.

The general error causes in clause 16.1.2 apply.

EXAMPLE 1: Using lengthof for values

```
lengthof('010'B)    // returns 3

lengthof('F3'H)     // returns 2

lengthof('F2'O)     // returns 1

lengthof (universal charstring : "Length_of_Example") // returns 17

// Given
type record length(0..10) of integer MyList;
var MyList MyListVar := { 0, 1, -, 2, - };

lengthof(MyListVar);
// returns 4 without respect to the fact, that the element MyListVar[2] is not initialized
```

EXAMPLE 2: Using lengthof for string-type templates

```
lengthof(charstring : "HELLO")          // returns 5

lengthof(octetstring : ('12'O, '34'O))  // returns 1

lengthof('1??1'B)                       // returns 4

lengthof(universal charstring : ? length(8)) // returns 8

lengthof('1*F'H)                        // shall cause an error

lengthof('1*F'H length (8))             // returns 8

lengthof(bitstring : ? length(2..infinity)) // shall cause an error

lengthof('00*FF'O length(1..2))         // returns 2

lengthof('1*49'H length(1..2))      // shall cause an error

lengthof('1'B length(3))                // shall cause an error

lengthof('1*1'B length(10..20))         // shall cause an error
```

EXAMPLE 3:

```
type record of integer RoI;
template RoI tr_roI1 := { 1, permutation(2, 3), ? }
template RoI tr_roI2 := {1, *, (2, 3) }
template RoI tr_roI3 := { 1, *, 10 } length(5)
template RoI tr_roI4 := { 1, 2, 3, * } length(1..2)
template RoI tr_roI5 := { 1, 2, 3, * } length(1..3)

lengthof (tr_roI1) // returns 4

lengthof (tr_roI2) // shall cause an error

lengthof (tr_roI3) // returns 5
```

```
lengthof (tr_roI4) // shall cause an error

lengthof (tr_roI5) // returns 3
```

C.30 Number of elements in a structured value

```
sizeof(in template (present) any_record_set_type inpar) return integer
```

This function returns the actual number of elements of a value or template of a **record** or **set** type (see note).

The function **sizeof** is applicable to templates of record and set types. The function is applicable only if the **sizeof** function gives the same result on all values that match the template.

NOTE: Only elements of the TTCN-3 object, which is the parameter of the function are calculated; i.e. no elements of nested types/values are taken into account at determining the return value.

In addition to the general error causes in clause 16.1.2, error causes are:

- when inpar is a template and it can match values of different sizes.

EXAMPLE:

```
// Given
type record MyPDU
    {   boolean field1  optional,
        integer field2
    };

template MyPDU MyTemplate
    { field1 := omit,
      field2 := 5
    };

sizeof(MyTemplate); // returns 1

type set S {
  integer f1,
  bitstring f2 optional,
  charstring f3 optional
}

template S tr_S1 := { f1 := (0..99), f2 := omit, f3 := ? }
template S tr_S2 := { f3 := *, f1 := 1, f2 := '00'B ifpresent }
template S tr_S3 := ({ f1 := 1, f2 := omit, f3 := "ABC" }, { f1 := 2, f3 := omit, f2 := '1'B })
template S tr_S4 := ?

sizeof(tr_S1) // returns 2
sizeof(tr_S2) // shall cause an error
sizeof(tr_S3) // returns 2
sizeof(tr_S4) // shall cause an error
```

C.31 The IsPresent function

```
ispresent(in template any_ type inpar) return boolean
```

This function is allowed for templates of all data types and returns:

- the value **true** if the given template fulfils the (present) template restriction as described in clause 15.8,
- the value **false** otherwise if no error is caused (see below).

NOTE 1: When the argument of **ispresent** is a subfield of a template field to which the "?" (*AnyValue*) matching is assigned, the extension mechanism specified in clause 15.6.2 applies.

NOTE 2: This means that whenever **ispresent**(MyTemplate) returns true,
- MyTemplate can safely be assigned to a non-optional field of the type of the template in a template variable
- MyTemplate can safely be used as an actual template(present) parameter or assigned to a variable of kind template(present).

In addition to the general error causes in clause 16.1.2, error causes are:

- inpar is referring to a field that is not accessible, e.g. embedded in a template or in a field using **omit** or "*" (*AnyValueOrNone*). Note, that this rule applies for any levels of embedding.

EXAMPLE:

```
// Given
type record MyRecord
  {
    record {
      boolean innerField1 optional,
      integer innerField2 optional
    } field1 optional,
        integer field2
    }

var MyRecord vl_MyRecord := { field1 := {}, field2 := 5 }

ispresent(vl_MyRecord.field1) // returns true

vl_MyRecord.field1 := omit

ispresent(vl_MyRecord.field1) // returns false

ispresent(vl_MyRecord.field1.innerField1) // shall return false because field1 is omitted

var template MyRecord vlt_MyRecord := { field1 := ?, field2 := 5 }

ispresent(vlt_MyRecord.field1) // returns true

ispresent(vlt_MyRecord.field1.innerField1) // shall cause an error because field1 is AnyValue
// (pls. note, that at expansion of field1 the optional field innerField1 obtains "*"
// that can match both a present and an omitted field

type record R { integer f1 optional, integer f2 optional }
template R t1 := {f1 := 1, f2 :=(2 .. 4) }
template R t2 := { f1 := omit, f2 := (5, 7) ifpresent }
template R t3 := {f1 := *, f2 :=? }

ispresent(t1.f1) // returns true

ispresent(t1.f2) // returns true

ispresent(t2.f1) // returns false

ispresent(t2.f2) // returns false

ispresent(t3.f1) // returns false

ispresent(t3.f2) // returns true
```

C.32 The IsChosen function

```
ischosen(in template any_union_type inpar) return boolean
```

This function returns the value **true** if and only if the data object reference specifies the variant of the **union** type that is actually selected for a given data object.

The function **ischosen** is applicable to templates of union types containing a specific value or a value list. It returns **true** if all the values matched by inpar have the given field selected. The result is **false** if there is another field of the union type on which **ischosen** would return true.

In addition to the general error causes in clause 16.1.2, error causes are:

- inpar is referring to a field that is not accessible, e.g. embedded in a template or in a field using **omit**, "?" (*AnyValue*) or "*" (*AnyValueOrNone*). Note, that this rule apply for any levels of embedding;
- when inpar is a template and it can match values containing different selected fields.

EXAMPLE 1:

```
type union U { integer f1, octetstring f2 }
template U t_U1 := {f1 := 1}
template U t_U2 := {f2 := ?}
template U t_U3 := ?
template U t_U4 := ({ f1 := 2 }, {f2 := 'AB'O })
template U t_U5 := ({ f2 := '12?'O }, { f2 := '*34'O length(2) })

ischosen(t_U1.f1) // returns true

ischosen(t_U1.f2) // returns false

ischosen(t_U2.f1) // returns false

ischosen(t_U2.f2) // returns true

ischosen(t_U3.f1) // shall cause an error

ischosen(t_U3.f2) // shall cause an error

ischosen(t_U4.f1) // shall cause an error

ischosen(t_U4.f2) // shall cause an error

ischosen(t_U5.f1) // returns false

ischosen(t_U5.f2) // returns true
```

EXAMPLE 2:

```
// Given
type union MyUnion
    {   PDU_type1   p1,
        PDU_type2   p2,
        PDU_type    p3
    }

// and given that MyPDU is a template of MyUnion type
// and received_PDU is also of MyUnion type
// then
MyPort.receive(MyPDU) -> value received_PDU
ischosen(received_PDU.p2)
// returns true if the actual instance of MyPDU carries a PDU of the type PDU_type2
```

C.33 The Regexp function

```
regexp(
  in template (value) any_character_string_type inpar,
  in template (present) any_character_string_type expression,
  in integer groupno
) return any_character_string_type
```

This function returns the substring of the input character string inpar, which is the content of n-th group matching to the expression. The parameters inpar and expression shall be a value or a template of **charstring** or **universal charstring** types. In case inpar is a template, it shall contain the specific value matching mechanism only. The type of expression shall be **universal charstring** only when the type of inpar is **universal charstring.** When expression is a template it shall contain the specific value or pattern matching mechanisms only. The parameter groupno shall be a non-negative integer. The type of the character string returned is the root type of inpar.

First inpar (or in case inpar is a template, its value equivalent) shall be matched against expression. If expression is not a template containing a pattern matching mechanism, it shall be processed by this predefined function as if it was a character pattern as described in clause B.1.5. If this matching is unsuccessful, an empty string shall be returned. If this matching is successful, the substring of inpar shall be returned, which matched the groupno-s group of expression during the matching. Group numbers are assigned by the order of occurrences of the opening bracket of a group and counted starting from 0 by step 1.

In addition to the general error causes in clause 16.1.2, error causes are:

- when inpar is a template, it contains other matching mechanism than specific value or character pattern;
- when expression is a template, it contains other matching mechanism than specific value or character pattern;
- inpar is of charstring type and expression is of universal charstring type;
- groupno is a negative integer;
- there is no groupno -s group in expression.

EXAMPLE:

```
// Given
var charstring myInput := "   simple text for a regexp example   ";
var charstring myString;

myString := regexp(myInput,charstring:"?+(text)?+",0) //will return "text"

myString := regexp(myInput,charstring:"?+(text)?+",1) //causes an error as there is
                                                      //no group with index 1
myString := regexp(myInput,charstring:"(?+)(text)(?+)",0) //will return "   simple "

myString := regexp(myInput,charstring:"(?+)(text)(?+)",2) //will return
                                                        //" for a regexp example   "
myString := regexp(myInput,charstring:"((?+)(text)(?+))",0) //will return the whole inpar,
                                                  //i.e. "   simple text for a regexp example   "
myString := regexp(myInput,charstring:"(([ ]+)(text)(?+))",0) //will return an empty string
                                                    //as expression does not matches inpar
myString := regexp(myInput,universal charstring:"?+(text)?+",0) //will cause an error as
                                                    // inpar is of type charstring, while
                                                    // expression is of type universal charstring

myInput := "      date: 2001-10-20 ;  msgno: 17; exp  "
var template charstring myPattern := pattern"([ /t]#(,)date:[ \d\-]#(,);[ /t]#(,)msgno: (\d#(1,3)); (exp)#(0,1))"
//please note, that only the very first opening bracket and the bracket before "\d" denotes
// groups; "#(,)", "#(1,3)" and "#(0,1)" denotes matching the preceding expression several time


myString := regexp(myInput, myPattern,1) //will return the value "17".

//An example of a wrapper function to count groups from 1 and return the complete p_inpar
//if p_groupno equals 0
function regexp0(
  in template charstring p_inpar,
  in template charstring p_expression,
  in integer p_groupno)
return charstring {
    var template charstring extended_expr := pattern "({p expression})";
    return regexp(p inpar, extended_expr, p_groupno )
}
```

C.34 The Substring function

```
substr(
  in template (present) any_string_or_sequence_type inpar,
  in integer index,
  in integer count
) return input_string_or_sequence_type
```

This function returns a substring or subsequence from a value that is of a binary string type (**bitstring**, **hexstring**, **octetstring)** , a character string type (**charstring**, **universal charstring**), or a sequence type (**record of**, **set of** or array). The type of the substring or subsequence returned is the root type of the input parameter. The starting point of substring or subsequence to return is defined by the second parameter (`index`). Indexing starts from zero. The third input parameter (`count`) defines the length of the substring or subsequence to be returned. The units of length for string types are as defined in table 4 of the present document. For sequence types, the unit of length is element.

NOTE: Please note that the root types of arrays is **record of**, therefore if `inpar` is an array the returned type is **record of**. This, in same cases, may lead to different indexing in `inpar` and in the returned value.

When used on templates of character string types, only the inside matching mechanisms *AnyElement* and *AnyElementsOrNone* are allowed in `inpar` and the function shall return the character representation of the matching mechanisms, i.e. "?" for *AnyElement* and "*" for *AnyElementsOrNone*. When inpar is a template of binary string or sequence type or is an array, only the specificvalue and *AnyElement* matching mechanisms are allowed and the substring or subsequence to be returned shall not contain *AnyElement.*

In addition to the general error causes in clause 16.1.2, error causes are:

- `index` is less than zero;
- `count` is less than zero;
- `index+count` is greater than **lengthof**(`inpar`);
- `inpar` is a template of a character string type and contains a matching mechanism other than *AnyElement* or *AnyElementsOrNone*;
- `inpar` is a template of a binary string or sequence type or array and it contains other matching mechanism as specific value and *AnyElement*;
- `inpar` is a template of a binary string or sequence type or array and the substring or subsequence to be returned contains the *AnyElement* matching mechanism.

EXAMPLE:

```
substr('00100110'B, 3, 4)       // returns '0011'B

substr('ABCDEF'H, 2, 3)     // returns 'CDE'H

substr('01AB23CD'O, 1, 2)       // returns 'AB23'O

substr("My name is JJ", 11, 2)  // returns "JJ"

substr({ 4, 5, 6 }, 1, 2)       // returns {5, 6}
```

C.35 The Replace function

```
replace(
  in any_string_or_sequence_type inpar,
  in integer index,
  in integer len,
  in any_string_or_sequence_type repl
) return any_string_or_sequence type
```

This function replaces the substring or subsequence of value `inpar` at index `index` of length `len` with the string or sequence value `repl` and returns the resulting string or sequence. `inpar`shall not be modified. If `len` is `0` the string or sequence `repl` is inserted. If `index` is `0`, `repl` is inserted at the beginning of `inpar`. If `index` is **lengthof**(`inpar`), `repl` is inserted at the end of `inpar`. `inpar`and `repl`, and the returned string or sequence shall be of the same root type. The function replace can be applied to **bitstring**, **hexstring**, **octetstring**, or any character string, **record of**, **set of**, or arrays. Note that indexing in strings starts from zero.

NOTE: Please note that the root types of arrays is **record of**, therefore if inpar or repl or both are an array, the returned type is **record of**. This, in same cases, may lead to different indexing in inpar and/or repl and in the returned value.

In addition to the general error causes in clause 16.1.2, error causes are:

- inpar or repl are not of string, **record of**, **set of**, or array type;
- inpar and repl are of different root type;
- index is less than 0 or greater than **lengthof**(inpar);
- len is less than 0 or greater than **lengthof**(inpar);
- index+len is greater than **lengthof**(inpar).

EXAMPLE:

```
replace ('00000110'B, 1, 3, '111'B) // returns '01110110'B
replace ('ABCDEF'H, 0, 2, '123'H)       // returns '123CDEF'H
replace ('01AB23CD'O, 2, 1, 'FF96'O)    // returns '01ABFF96CD'O
replace ("My name is JJ", 11, 1, "xx")  // returns "My name is xxJ"
replace ("My name is JJ", 11, 0, "xx")  // returns "My name is xxJJ"
replace ("My name is JJ", 2, 2, "x")    // returns "Myxame is JJ",
replace ("My name is JJ", 12, 2, "xx")  // produces test case error
replace ("My name is JJ", 13, 2, "xx")  // produces test case error
replace ("My name is JJ", 13, 0, "xx")  // returns "My name is JJxx"
```

C.36 The random number generator function

```
rnd([in float seed]) return float
```

The **rnd** function returns a (pseudo) random number less than 1 but greater or equal to 0. The random number generator is initialized by means of an optional seed value (a numerical float value). If no new seed is provided, the last generated number will be used as seed for the next random number. Without a previous initialization a value calculated from the system time will be used as seed value when **rnd** is used the first time.

Each time the **rnd** function is initialized with the same seed value, it shall repeat the same sequence of random numbers.

To produce a random integers in a given range, the following formula can be used:

```
float2int(int2float(upperbound - lowerbound +1)*rnd()) + lowerbound
// Here, upperbound and lowerbound denote highest and lowest number in range.
```

In addition to the general error causes in clause 16.1.2, error causes are:

- seed is **infinity**, **-infinity** or **not_a_number**.

C.37 Enumerated to integer

```
enum2int(in Enumerated_type inpar) return integer
```

This function accepts an enumeration value and returns the **integer** value associated to the enumeration (see also clause 6.2.4).

The general error causes in clause 16.1.2 apply.

EXAMPLE:

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

type enumerated MySecondEnumType {
    Saturday(-3), Sunday (0), Monday
};

//within a dynamic language element:
var MyFirstEnumType vl_FirstEnum := Monday;
var MySecondEnumType vl_SecondEnum := Monday;

enum2int(vl_FirstEnum) // returns 0
enum2int(vl_SecondEnum) // returns 1

vl_FirstEnum := Wednesday;
vl_SecondEnum := Saturday;
enum2int(vl_FirstEnum) // returns 2
enum2int(vl_SecondEnum) // returns -3

vl_FirstEnum := Friday;
vl_SecondEnum := Sunday;
enum2int(vl_FirstEnum) // returns 4
enum2int(vl_SecondEnum) // returns 0
```

C.38 The IsValue function

```
isvalue(in template any_type inpar) return boolean;
```

The function shall accept templates of any known type. The function shall return **true**, if inpar is completely initialized and resolves to a specific value. If inpar is of a structured type or array, **omit** is considered to be a concrete value for optional fields, i.e. the function shall also return true if optional fields of inpar are set to omit. The function shall return **false** otherwise.

If the isvalue function is used with a non-selected choice of a union type value or template, this shall cause an error.

The **null** value assigned to default and component references shall be considered as concrete values.

In addition to the general error causes in clause 16.1.2, error causes are:

- inpar is referring to a field that is not accessible, e.g. embedded in a template or in a template field using **omit** or "*" (*AnyValueOrNone*). Note that this rule applies for any levels of embedding.

EXAMPLE 1: Simple types

```
template charstring ts_char0 := "ABCD"; //template containing a specific value matching
template charstring tr_char1 := "AB?D"; //template containing a specific value matching
                                        //note, that "?" is not a matching symbol in this case
template charstring tr_char2 := pattern "ABCD"; //a pattern matching a single value only
template charstring tr_char3 := pattern "AB?D"; //pattern matching
template charstring tr_char4 := ("ABCD"); // template containing a specific value (expression)
template charstring tr_char5 := ("ABCD","EFGH"); //a value list matching a single value only

isvalue(ts_char0); // shall return true
isvalue(tr_char1); // shall return true
isvalue(tr_char2); // shall return false
isvalue(tr_char3); // shall return false
isvalue(tr_char4); // shall return true similarly to e.g. isvalue((2)) shall return true
isvalue(tr_char5); // shall return false
```

EXAMPLE 2: Special types

```
var default vl_default := null;
isvalue(vl_default); // shall return true
```

EXAMPLE 3: Record/set types

```
type record MyRec {
    integer f1 optional,
    integer f2 optional
}

var MyRec vl_MyRec;
var template MyRec vlt_MyRec;

isvalue(vl_MyRec); // shall return false
isvalue(vlt_MyRec); // shall return false

vl_MyRec  := { f1 := 5, f2 := omit }
vlt_MyRec := { f1 := ?, f2 := 5 }

isvalue(vl_MyRec); // shall return true
isvalue(vl_MyRec.f2); // shall return false;
isvalue(vlt_MyRec); // shall return false
isvalue(vlt_MyRec.f1); // shall return false
isvalue(vlt_MyRec.f2); // shall return true

vlt_MyRec.f2 := omit;

isvalue(vlt_MyRec.f2); // shall return false
```

EXAMPLE 4: Union types

```
type union MyUnion {
    integer ch1,
    integer ch2
}

template MyUnion ts_MyUnion := { ch1 := 5 }
template MyUnion tr_MyUnion := { ch1 := ? }

var MyUnion vl_ MyUnion;

isvalue(ts_MyUnion); // shall return true
isvalue(tr_MyUnion); // shall return false
isvalue(tr_MyUnion.ch1); // shall return false;
// note, this is different from ischosen(tr_MyUnion.ch1) as isvalue checks the content of the
// choice ch1, while ischosen is checking if ch1 has been selected or not
isvalue(tr_MyUnion.ch2); // shall cause an error;
```

C.39 The encoding function

```
encvalue(in template (value) any_type inpar) return bitstring
```

The **encvalue** function encodes a value or template into a bitstring. When the actual parameter that is passed to `inpar` is a template, it shall resolve to a specific value (the same restrictions apply as for the argument of the **send** statement). The returned bitstring represents the encoded value of `inpar`, however, the TTCN-3 test system need not to make any check on its correctness.

In addition to the general error causes in clause 16.1.2, error causes are:

- Encoding fails due to a runtime system problem (i.e. no encoding function exists for the actual type of `inpar`).

C.40 The decoding function

```
decvalue(inout bitstring encoded_value, out any_type decoded_value) return integer
```

The **decvalue** function decodes a bitstring into a value. The test system shall suppose that the bitstring `encoded_value` represents an encoded instance of the actual type of `decoded_value`.

If the decoding was successful, then the used bits are removed from the parameter `encoded_value`, the rest is returned (in the parameter `encoded_value`), and the decoded value is returned in the parameter `decoded_value`.

If the decoding was unsuccessful, the actual parameters for `encoded_value` and `decoded_value` are not changed. The function shall return an integer value to indicate success or failure of the decoding below:

- The return value 0 indicates that decoding was successful.
- The return value 1 indicates an unspecified cause of decoding failure.
- The return value 2 indicates that decoding could not be completed as `encoded_value` did not contain enough bits.

The restrictions in clause 16.1.2 apply. If any of these restrictions is applicable, the return value shall be 1.

C.41 The testcasename function

```
testcasename () return charstring
```

The **`testcasename`** function shall return the unqualified name of the actually executing test case.

EXAMPLE 1:

```
module MyTCModule {
    :
    testcase MyTestCase1 () runs on MTC system TSI
    {
        var charstring v_TCname := testcasename ();
          // will return the charstring "MyTestCase1"
        :
    }
    :
    testcase MyTestCase2 () runs on MTC system TSI
    {
        :
    }
    :
}
module MyTSModule {
    :
    function MyStartAPTC() runs on PTC {
        var charstring v_TCname := testcasename ();
          // will return charstring "MyTestCase1", if the function is
          // called by a test component during the execution of MyTestCase1
          // will return charstring "MyTestCase2", if the function is
          // called by a test component when MyTestCase2 is being executed
    }
    :
}
```

When the function **`testcasename`** is called if the control part is being executed but no testcase, it shall return the empty string.

EXAMPLE 2:

```
module MyModule {
    :
    control
    {
        var charstring v_TCname := testcasename () // will return charstring ""
        :
    }
    :
}
```

The general error causes in clause 16.1.2 apply.

C.42 Integer to enumerated

```
int2enum ( in integer inpar, out Enumerated_type outpar)
```

This function converts an integer value into a value of a given enumerated type. The integer value shall be provided as in parameter and the result of the conversion shall be stored in an out parameter. The type of the out parameter determines the type into which the in parameter is converted.

The general error causes in clause 16.1.2 apply.

EXAMPLE:

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

type enumerated MySecondEnumType {
    Saturday(-3), Sunday (0), Monday
};

//within a dynamic language element:
var MyFirstEnumType firstEnum := Tuesday;
var MySecondEnumType secondEnum := Sunday;

int2enum(0, firstEnum)  // firstEnum == Monday
int2enum(1, secondEnum) // secondEnum == Monday
```

Annex D (normative): Preprocessing macros

This annex defines a set of preprocessing macros. A preprocessing macro is a macro that is replaced by a preprocessor or a compiler with a **charstring** or **integer** value respectively before compilation. Preprocessing macros shall not be replaced inside literal **charstring** values and templates and not in TTCN-3 comments. In the TTCN-3 code, it can be used like a **charstring** or an **integer** value respectively.

D.1 Preprocessing macro __MODULE__

The **__MODULE__** preprocessing macro denotes the module name in which the macro is used. A preprocessor or compiler shall replace all occurrences of **__MODULE__** with the actual module name in form of a **charstring** value.

D.2 Preprocessing macro __FILE__

The **__FILE__** preprocessing macro denotes the canonical (absolute) file name, i.e. the full path and the basic file name, in which the macro is used. A preprocessor or compiler shall replace all occurrences of **__FILE__** with the actual canonical (absolute) file name in form of a **charstring** value.

NOTE: The format of the canonical file name depends on the operating system and is not specified by the present document.

EXAMPLE:

```
const charstring MyConst:= __FILE__;
//MyConst is for example "/home/myhome/MyTest.ttcn"
```

D.3 Preprocessing macro __BFILE__

The **__BFILE__** preprocessing macro denotes the basic (relative) file name, i.e. without path, in which the macro is used. A preprocessor or compiler shall replace all occurrences of **__BFILE__** with the actual basic (relative) file name in form of a **charstring** value.

NOTE: The format of the basic file name depends on the operating system and is not specified by the present document.

EXAMPLE:

```
const charstring MyConst:= __BFILE__;
// MyConst is for example "MyTest.ttcn"
```

D.4 Preprocessing macro __LINE__

The **__LINE__** preprocessing macro denotes the line number of the file in which the macro is used. A preprocessor or compiler shall replace each occurrence of **__LINE__** with the actual line number in form of an **integer** value.

A file starts with line number **1**. Each newline shall increase the line number by **1** (see clause A.1.5.1). Also newlines of commented lines shall increase the line number by **1**.

D.5 Preprocessing macro __SCOPE__

The **__SCOPE__** preprocessing macro denotes the unqualified name of the lowest named basic scope unit in which the macro is used. According to clause 5.2, basic scope units of TTCN-3 are module definitions part, module control part, component types, functions, altsteps, test cases, statement blocks, templates and user defined named types. Statement blocks have no name and therefore, a **__SCOPE__** preprocessing macro used in a statement block refers to the next higher named basic scope unit.

A preprocessor or compiler shall replace all occurrences of **__SCOPE__** with a **charstring** value which includes:

a) the module name, if the lowest named scope unit is the module definitions part;

b) "**control**", if the lowest named scope unit is the module control part;

c) a component type name, if the lowest named scope unit is a component type definition;

d) a test case name, if the lowest named scope unit is a test case definition;

e) an altstep name, if the lowest named scope is an altstep definition;

f) a function name, if the lowest named scope is a function definition;

g) a template name, if the lowest named scope is a template definition (local or global); or

h) the type name, if the lowest named scope is a user defined named type definition.

NOTE: The **__SCOPE__** preprocessing macro cannot be used to retrieve the names of other kinds of definitions, like for example names of groups of definitions or names of global constants.

EXAMPLE 1: Using __SCOPE__ in constant and template definitions

```
module MyModule
{
    const charstring MyConst := __SCOPE__;          // MyConst contains "MyModule"
    template charstring MyTemplate := __SCOPE__;    // MyTemplate contains "MyTemplate"

type record MyRecord1
{
    charstring field11,
    charstring field12
}

template MyRecord1 MyTemplate1 (charstring p := __SCOPE__) :=
{
    field11 := p,
    field12 := __SCOPE__                            // field12 contains "MyTemplate1"
}

function MyFunction() {
  var template MyRecord1 v_Myvar1 := MyTemplate1;
    // field11 of MyTemplate1 will contain the default value of parameter p,
    // i.e. "MyTemplate1"
};
}
```

EXAMPLE 2: Using __SCOPE__ in a structured type scope

```
type record MyRecord2 {
    charstring field21,
    charstring field22 ("a", "b", __SCOPE__)
    // list constrained field: a legal values are "a", "b" or "MyRecord2"
}

template MyRecord2 MyTemplate2 := {
    field21 := "a",
    field22 := "MyRecord2"                          // a valid specific value matching
}

template MyRecord2 MyTemplate3 := {
    field21 := "a",
    field22 := __SCOPE__
```

```
    // Causes an error as __SCOPE__ is replaced with "MyTemplate3",
    // which is violating the list constraint of field22
}
```

EXAMPLE 3: Using __SCOPE__ in an embedded structured type scope

```
type record MyRecord3 {
    charstring field31,
    record {
        charstring field321 ("a", "b", __SCOPE__)
        // list constrained field: a legal value shall be "a", "b" or "MyRecord3"
    } field32
}

template MyRecord3 MyTemplate4 :=
{
    field31 := "a",
    field32 :=
    {
        field321 := "MyRecord3"                    // a valid specific value matching
    }
}

template MyRecord3 MyTemplate5 :=
{
    field31 := "a",
    field32 :=
    {
        field321 := __SCOPE__
        // Causes and error as __SCOPE__ is replaced with "MyTemplate5",
        // which is violating the list constraint of field321
    }
}
```

Annex E (informative): Library of Useful Types

E.1 Limitations

Names of types added to this library are to be unique within the whole language and within the library (i.e. are not to be one of the names defined in annex C). Names defined in this library are not to be used by TTCN-3 users as identifiers of other definitions than given in this annex.

NOTE: Therefore type definitions given in this annex may be repeated in TTCN-3 modules but no type distinct from the one specified in this annex can be defined with one of the identifiers used in this annex.

E.2 Useful TTCN-3 types

E.2.1 Useful simple basic types

E.2.1.0 Signed and unsigned single byte integers

These types support integer values of the range from -128 to 127 for the signed and from 0 to 255 for the unsigned type. The value notation for these types is the same as the value notation for the integer type. Values of these types are to be encoded and decoded as they were represented on a single byte within the system independently from the actual representation form used.

NOTE: Encoding of values of these types may be the same or may differ from each other and from the encoding of the integer type (the root type of these useful types) depending on the actual encoding rules used. Details of encoding rules are out of the scope of the present document.

Type definitions for these types are:

```
type integer    byte           (-128 .. 127)   with { variant "8 bit" };

type integer    unsignedbyte   (0 .. 255)      with { variant "unsigned 8 bit" };
```

E.2.1.1 Signed and unsigned short integers

These types support integer values of the range from -32 768 to 32 767 for the signed and from 0 to 65 535 for the unsigned type. The value notation for these types is the same as the value notation for the integer type. Values of these types are to be encoded and decoded as they were represented on two bytes within the system independently from the actual representation form used.

NOTE: Encoding of values of these types may be the same or may differ from each other and from the encoding of the integer type (the root type of these useful types) depending on the actual encoding rules used. Details of encoding rules are out of the scope of the present document.

Type definitions for these types are:

```
type integer    short          (-32768 .. 32767)   with { variant "16 bit" };

type integer    unsignedshort  (0 .. 65535)        with { variant "unsigned 16 bit" };
```

E.2.1.2 Signed and unsigned long integers

These types support integer values of the range from -2 147 483 648 to 2 147 483 647 for the signed and from 0 to 4 294 967 295 for the unsigned type. The value notation for these types is the same as the value notation for the integer type. Values of these types are to be encoded and decoded as they were represented on four bytes within the system independently from the actual representation form used.

NOTE: Encoding of values of these types may be the same or may differ from each other and from the encoding of the integer type (the root type of these useful types) depending on the actual encoding rules used. Details of encoding rules are out of the scope of the present document.

Type definitions for these types are:

```
type integer    long    (-2147483648 .. 2147483647)
                with { variant "32 bit" };

type integer    unsignedlong    (0 .. 4294967295)
                with { variant "unsigned 32 bit" };
```

E.2.1.3 Signed and unsigned longlong integers

These types support integer values of the range from -9 223 372 036 854 775 808 to 9 223 372 036 854 775 807 for the signed and from 0 to 18 446 744 073 709 551 615 for the unsigned type. The value notation for these types is the same as the value notation for the integer type. Values of these types are to be encoded and decoded as they were represented on eight bytes within the system independently from the actual representation form used.

NOTE: Encoding of values of these types may be the same or may differ from each other and from the encoding of the integer type (the root type of these useful types) depending on the actual encoding rules used. Details of encoding rules are out of the scope of the present document.

Type definitions for these types are:

```
type integer    longlong    (-9223372036854775808 .. 9223372036854775807)
                with { variant "64 bit" };

type integer    unsignedlonglong    (0 .. 18446744073709551615)
                with { variant "unsigned 64 bit" };
```

E.2.1.4 IEEE 754 floats

These types support the ANSI/IEEE 754 [6] for binary floating-point arithmetic. The type IEEE 754 [6] float supports floating-point numbers with base 10, exponent of size 8, mantissa of size 23 and a sign bit. The type IEEE 754 [6] double supports floating-point numbers with base 10, exponent of size 11, mantissa of size 52 and a sign bit. The type IEEE 754 [6] `extfloat` supports floating-point numbers with base 10, minimal exponent of size 11, minimal mantissa of size 32 and a sign bit. The type IEEE 754 [6] `extdouble` supports floating-point numbers with base 10, minimal exponent of size 15, minimal mantissa of size 64 and a sign bit.

Values of these types are to be encoded and decoded according to the IEEE 754 [6] definitions. The value notation for these types is the same as the value notation for the float type (base 10).

NOTE: Precise encoding of values of this type depends on the actual encoding rules used. Details of encoding rules are out of the scope of the present document.

Type definitions for these types are:

```
type    float   IEEE754float       with { variant "IEEE754 float" };

type    float   IEEE754double      with { variant "IEEE754 double" };

type    float   IEEE754extfloat    with { variant "IEEE754 extended float" };

type    float   IEEE754extdouble   with { variant "IEEE754 extended double" };
```

E.2.2 Useful character string types

E.2.2.0 UTF-8 character string "utf8string"

This type supports the whole character set of the TTCN-3 type **universal charstring** (see paragraph d) of clause 6.1.1). Its distinguished values are zero, one, or more characters from this set. Values of this type are entirely (e.g. each character of the value individually) to be encoded and decoded according to the UCS Transformation Format 8 (UTF-8) as defined in annex R of ISO/IEC 10646 [2]. The value notation for this type is the same as the value notation for the **universal charstring** type.

The type definition for this type is:

```
type universal charstring utf8string with { variant "UTF-8" };
```

E.2.2.1 BMP character string "bmpstring"

This type supports the Basic Multilingual Plane (BMP) character set of ISO/IEC 10646 [2]. The BMP represents all characters of plane 00 of group 00 of the Universal Multiple-octet coded Character Set. Its distinguished values are zero, one, or more characters from the BMP. Values of this type are entirely (e.g. each character of the value individually) to be encoded and decoded according to the UCS-2 coded representation form (see clause 14.1 of ISO/IEC 10646 [2]). The value notation for this type is the same as the value notation for the **universal charstring** type.

NOTE: The type "bmpstring" supports a subset of the TTCN-3 type **universal charstring.**

The type definition for this type is:

```
type universal charstring bmpstring ( char ( 0,0,0,0 ) .. char ( 0,0,255,255) )
with { variant "UCS-2" };
```

E.2.2.2 UTF-16 character string "utf16string"

This type supports all characters of planes 00 to 16 of group 00 of the Universal Multiple-octet coded Character Set (see ISO/IEC 10646 [2]). Its distinguished values are zero, one, or more characters from this set. Values of this type are entirely (e.g. each character of the value individually) to be encoded and decoded according to the UCS Transformation Format 16 (UTF-16) as defined in annex Q of ISO/IEC 10646 [2]. The value notation for this type is the same as the value notation for the **universal charstring** type.

NOTE: The type "utf16string" supports a subset of the TTCN-3 type **universal charstring.**

The type definition for this type is:

```
type universal charstring utf16string   ( char ( 0,0,0,0 ) .. char ( 0,16,255,255) )
with { variant "UTF-16" };
```

E.2.2.3 ISO/IEC 10646 character string "iso8859string"

This type supports all characters in all alphabets defined in the multiparty standard ISO/IEC 10646 [2]. Its distinguished values are zero, one, or more characters from the ISO/IEC 10646 [2] character set. Values of this type are entirely (e.g. each character of the value individually) to be encoded and decoded according to the coded representation as specified in ISO/IEC 10646 [2] (an 8-bit coding). The value notation for this type is the same as the value notation for the **universal charstring** type.

NOTE 1: The type "iso8859string" supports a subset of the TTCN-3 type **universal charstring.**

NOTE 2: In each ISO/IEC 10646 [2] alphabet the lower part of the character set table (positions 02/00 to 07/14) is compatible with the ITU-T Recommendation T.50 [4] character set. Hence all extra language specific characters are defined for the upper part of the character table only (positions 10/00 to 15/15).

The type definition for this type is:

```
type universal charstring iso8859string ( char ( 0,0,0,0 ) .. char ( 0,0,0,255) )
with { variant "8 bit" };
```

E.2.3 Useful structured types

E.2.3.0 Fixed-point decimal literal

This type supports the use of fixed-point decimal literal as defined in the IDL Syntax and Semantics version 2.6 [i.10]. It is specified by an integer part, a decimal point and a fraction part. The integer and fraction parts both consist of a sequence of decimal (base 10) digits. The number of digits is stored in "digits" and the size of the fraction part is given in "scale". The digits itself are stored in "value_". Value notation for this type is the same as the value notation for the record type. Values of this type are to be encoded and decoded as IDL fixed point decimal values.

NOTE: Precise encoding of values of this type depends on the actual encoding rules used. Details of encoding rules are out of the scope of the present document.

The type definition for this type is:

```
type record IDLfixed {
                        unsignedshort   digits,
                        short   scale,
                        charstring  value_
                    }
                    with { variant "IDL:fixed FORMAL/01-12-01 v.2.6" };
```

E.2.4 Useful atomic string types

E.2.4.1 Single ITU-T Recommendation T.50 character type

A type whose distinguished values are single characters of the version of ITU-T Recommendation T.50 [4] complying to the International Reference Version (IRV) as specified in clause 8.2 of ITU-T Recommendation T.50 [4] (see also note 1 to clause 6.1.1).

The type definition for this type is:

```
type charstring char646 length (1);
```

NOTE: The special string "8 bit" defined in clause 27.5 may be used with this type to specify a given encoding for its values. Also, other properties of the base type can be changed by using attribute mechanisms.

E.2.4.2 Single universal character type

A type whose distinguished values are single characters from ISO/IEC 10646 [2].

The type definition for this type is:

```
type universal charstring uchar length (1);
```

NOTE: Special strings defined in clause 27.5 except "8 bit" may be used with this type to specify a given encoding for its values. Also, other properties of the base type can be changed by using attribute mechanisms.

E.2.4.3 Single bit type

A type whose distinguished values are single binary digits.

The type definition for this type is:

```
type bitstring bit length (1);
```

E.2.4.4 Single hex type

A type whose distinguished values are single hexadecimal digits.

The type definition for this type is:

```
type hexstring hex length (1);
```

E.2.4.5 Single octet type

A type whose distinguished values are pairs of hexadecimal digits.

The type definition for this type is:

```
type octetstring octet length (1);
```

Annex F (informative): Operations on TTCN-3 active objects

This annex describes in a short form the semantics of operations on active objects in TTCN-3 being test components, timers and ports. This dynamic behaviour is written in the form of state machines with:

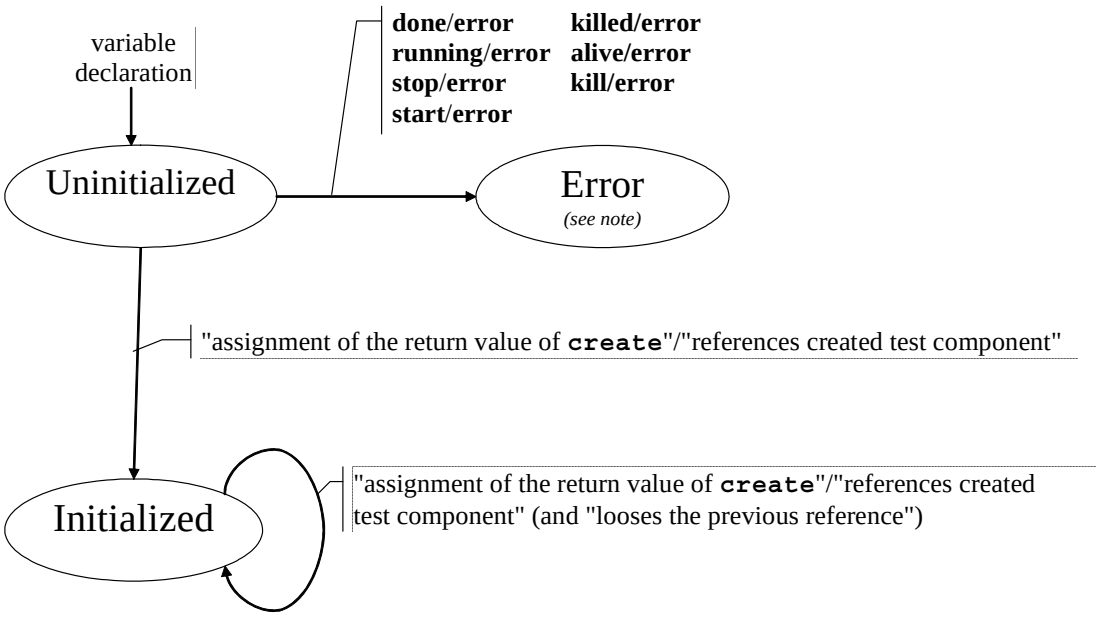
- the states being named and identified as nodes;
- the initial state being identified by an incoming arrow;
- transitions between states connecting two states (not necessarily different states) and identified as arrows;
- transitions being marked with the enabling condition for that transition (i.e. operation or statement calls) and the resulting condition (for example a test case error), both are separated by '/':
  - operation and statement calls are the TTCN-3 operations and statements applicable to the object (written in bold);
  - error as a resulting condition means testcase error (written in bold);
  - null as a resulting condition means that except of a possible state change no other results apply (written in bold);
  - match/no match refers to the matching result of a transition (written in bold);
  - concrete values are boolean or float results (written in bold italics);
  - all other resulting conditions are textually described (written in standard font);
- notes are used to explain further details of the state machine.

For further details, please refer to the operational semantics of TTCN-3 [1]. In case of any contradiction between this annex and the operational semantics of TTCN-3 [1] the latter takes precedence.

F.1 Test components

F.1.1 Test component references

Variables of test component types, the **`self`** and **`mtc`** operations are used to reference test components. The **`start`**, **`stop`**, **`done`** and **`running`** operations are not directly applied on test components but on component references. The test system has to decide if the operation requested should affect the component object itself or other action is appropriate (e.g. an error occurs when the reference of a stopped PTC is used in a component start operation). The **`create`** operation used to create PTCs returns a unique reference to the created PTC, which is typically bound to a test component variable. The behaviour related to test component variables themselves is shown in figure F.1.

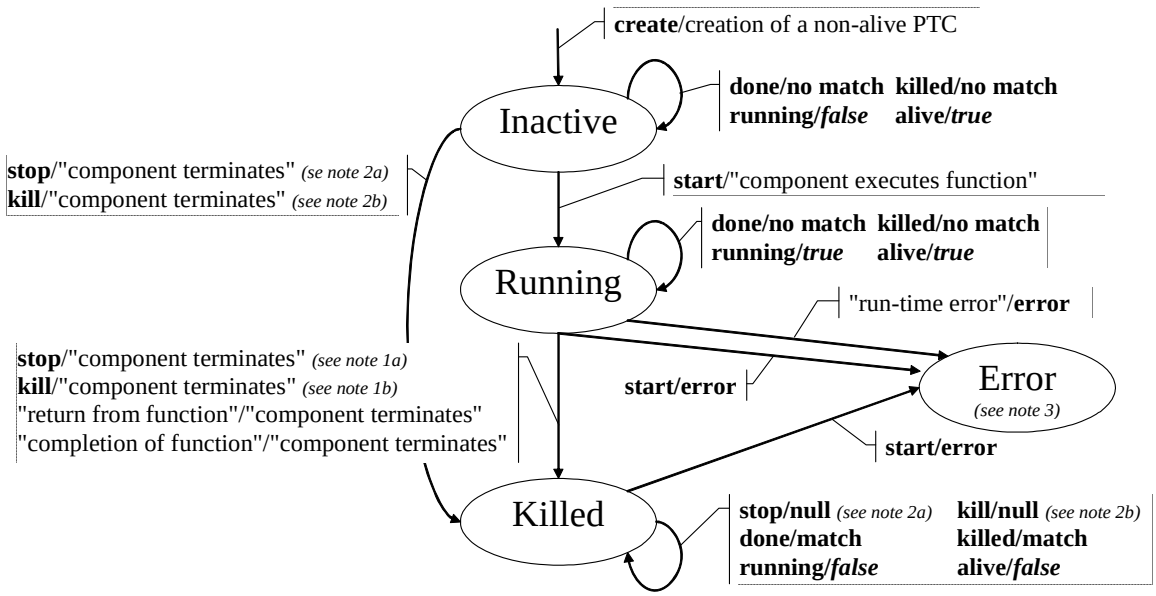


NOTE: Whenever a test component enters its error state, the error verdict is assigned to its local verdict, the test case terminates and the overall test case result will be error.

Figure F.1: Handling of test component references

F.1.2 Dynamic behaviour of PTCs

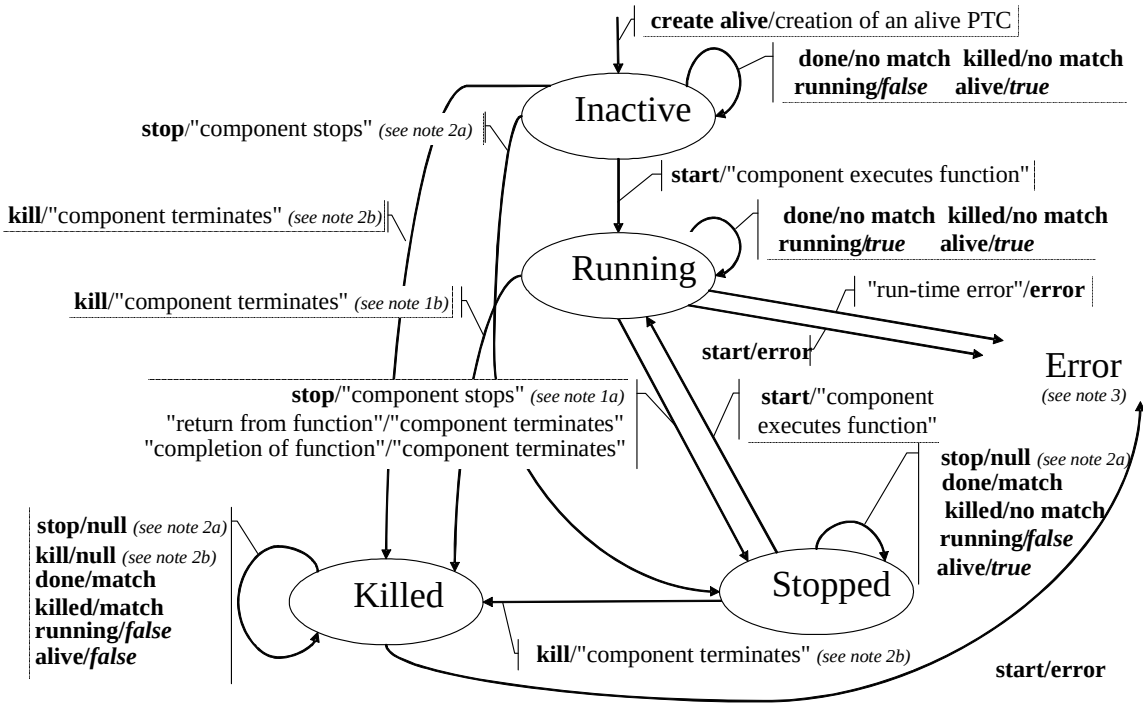
PTCs can be of non-alive type or alive-type. Non-alive type PTCs can be in Inactive, Running and Killed states. Their dynamic behaviour is shown in figure F.2.



NOTE 1: (a) Stop can be either a stop, self.stop or a stop from another test component.
(b) Kill can be either a kill, self.kill, a kill from another test component or a kill from the test system (in error cases).
NOTE 2: (a) Stop can be from another test component only.
(b) Kill can be from another test component or from the test system (in error cases) only.
NOTE 3: Whenever a test component enters its error state, the error verdict is assigned to its local verdict, the test case terminates and the overall test case result will be error.

Figure F.2: Dynamic behaviour of non-alive type PTCs

Alive-type PTCs can be in Inactive, Running, Stopped and Killed states. Their dynamic behaviour is shown in figure F.3.

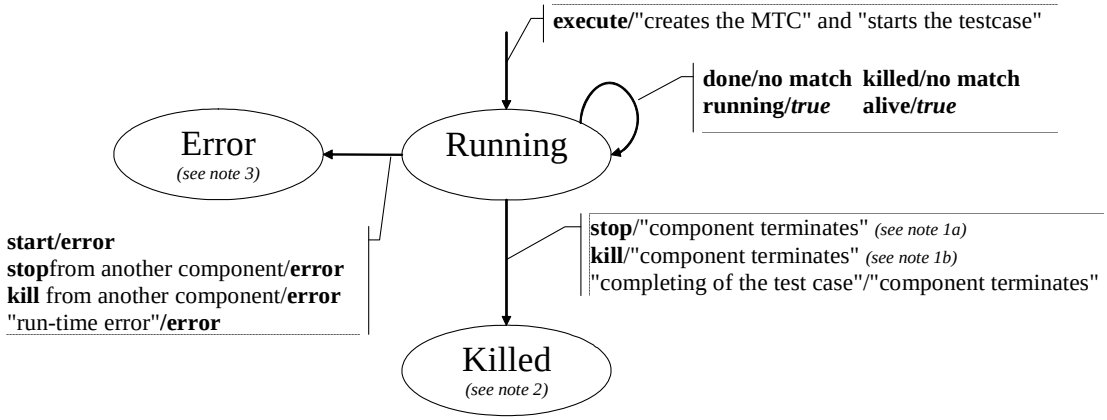


NOTE 1: (a) Stop can be either a stop, self.stop or a stop from another test component.
(b) Kill can be either a kill, self.kill, a kill from another test component or a kill from the test system (in error cases).
NOTE 2: (a) Stop can be from another test component only.
(b) Kill can be from another test component or from the test system (in error cases) only.
NOTE 3: Whenever a test component enters its error state, the error verdict is assigned to its local verdict, the test case terminates and the overall test case result will be error.

Figure F.3: Dynamic behaviour of alive-type PTCs

F.1.3 Dynamic behaviour of the MTC

The MTC can be in Running or Killed state. The dynamic behaviour of the MTC is shown in figure F.4.

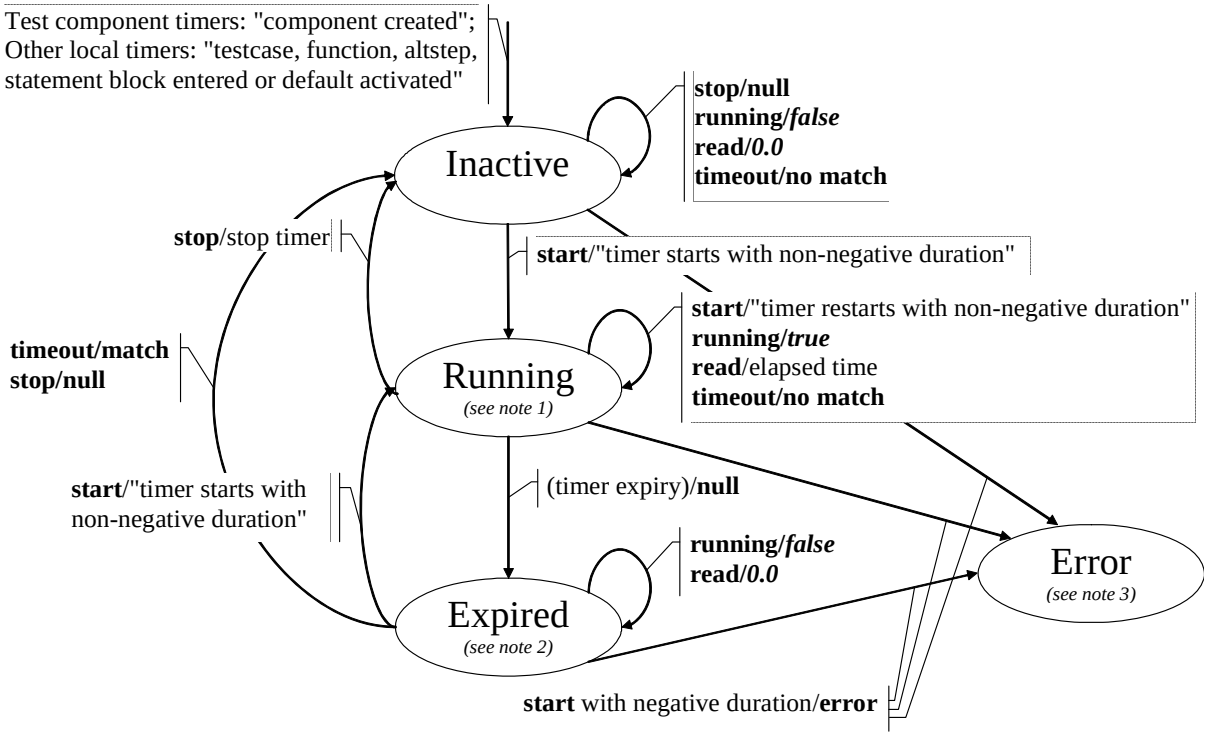


NOTE 1: (a) Stop can be either a stop, self.stop, a stop from another test component.
(b) Kill can be either a kill, self.kill, a kill from another test component or a kill from the test system (in error cases).
NOTE 2: All remaining PTCs are to be killed as well and the testcase terminates.
NOTE 3: Whenever the MTC enters its error state, the error verdict is assigned to its local verdict, the test case terminates and the overall test case result will be error.

Figure F.4: Dynamic behaviour of the MTC

F.2 Timers

Timers can be in Inactive, Running or Expired state. The dynamic behaviour of a timer is shown in figure F.5.



NOTE 1: For any scope unit, all timers in that scope being in Running state constitute the running-timer list.
NOTE 2: For any scope unit, all timers in that scope being in Expired state constitute the timeout-list.
NOTE 3: Whenever a timer enters its error state, the test component it belongs to enters also its error state, assigns a local error verdict, the test case terminates and the overall test case result will be error.

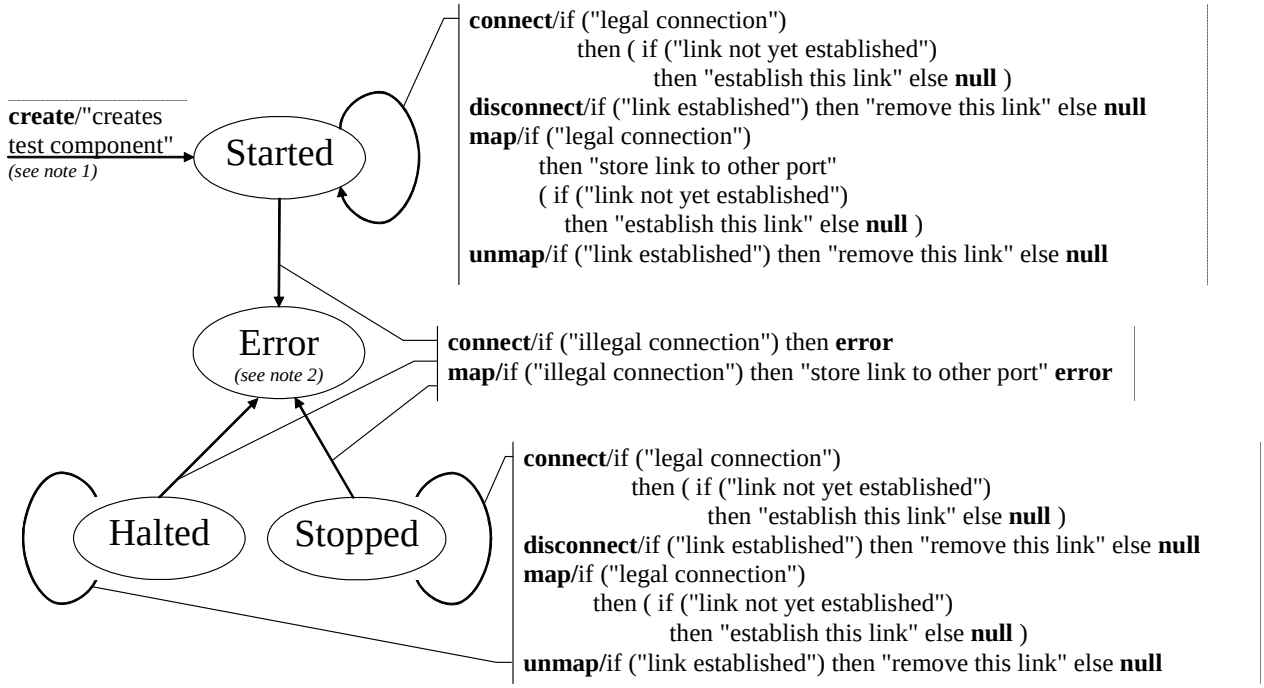
Figure F.5: Dynamic behaviour of timers

F.3 Ports

Ports can be in Started or Stopped state. As their behaviour is rather complex, the state machine has been split into a state machine giving the dynamic behaviour of configuration operations (i.e. connect, disconnect, map and unmap), of port controlling operations (i.e. start, stop and clear) and of communication operations (i.e. send, receive, call, getcall, raise, catch, reply, getreply and check). As trigger is a shorthand for an alt together with receive it is not considered here.

F.3.1 Configuration Operations

The port configuration operations (i.e. connect, disconnect, map and unmap) are indifferent to the state of the port. They show the behaviour shown in figure F.6.



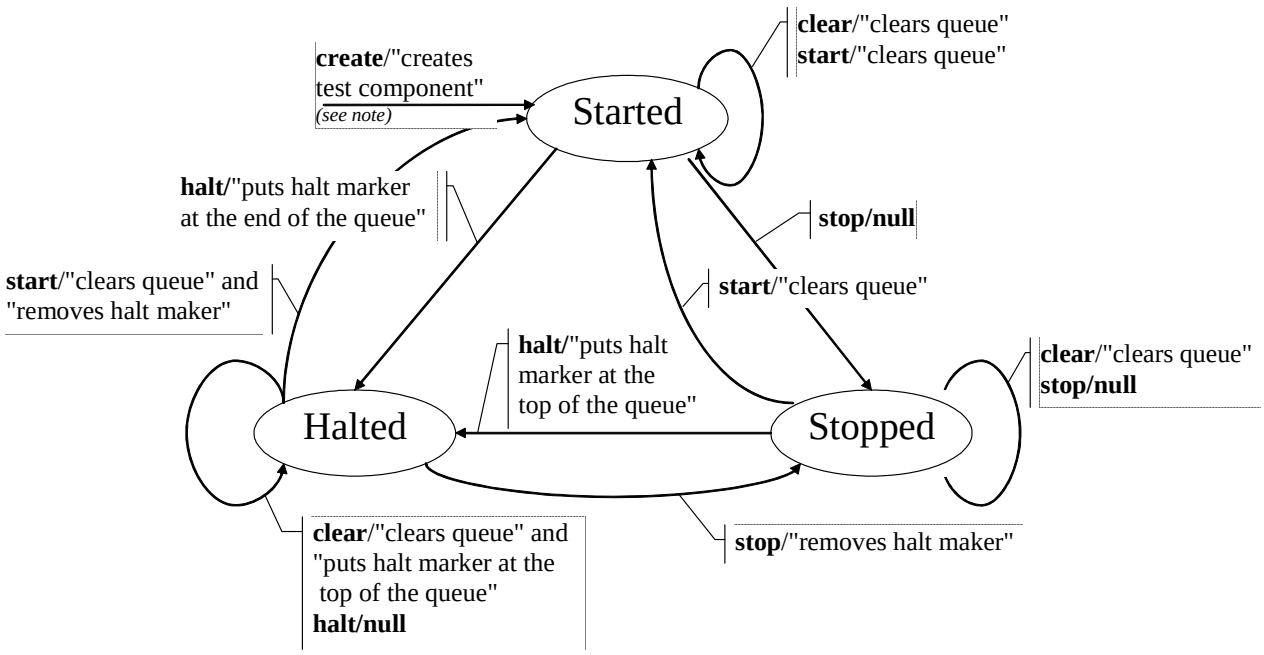
NOTE 1: When creating a PTC the ports of that PTC are created and started; when creating the MTC the ports of the MTC and the ports of the TSI are created and started.
NOTE 2: Whenever a port enters its error state, the test component it belongs to enters also its error state, assigns a local error verdict, the test case terminates and the overall test case result will be error.

Figure F.6: Dynamic behaviour of ports: port configuration operations

The transitions do not change the main state of the port, i.e. the port remains in the Started or Stopped state.

F.3.2 Port Controlling Operations

The results of port controlling operations are shown in figure F.7.

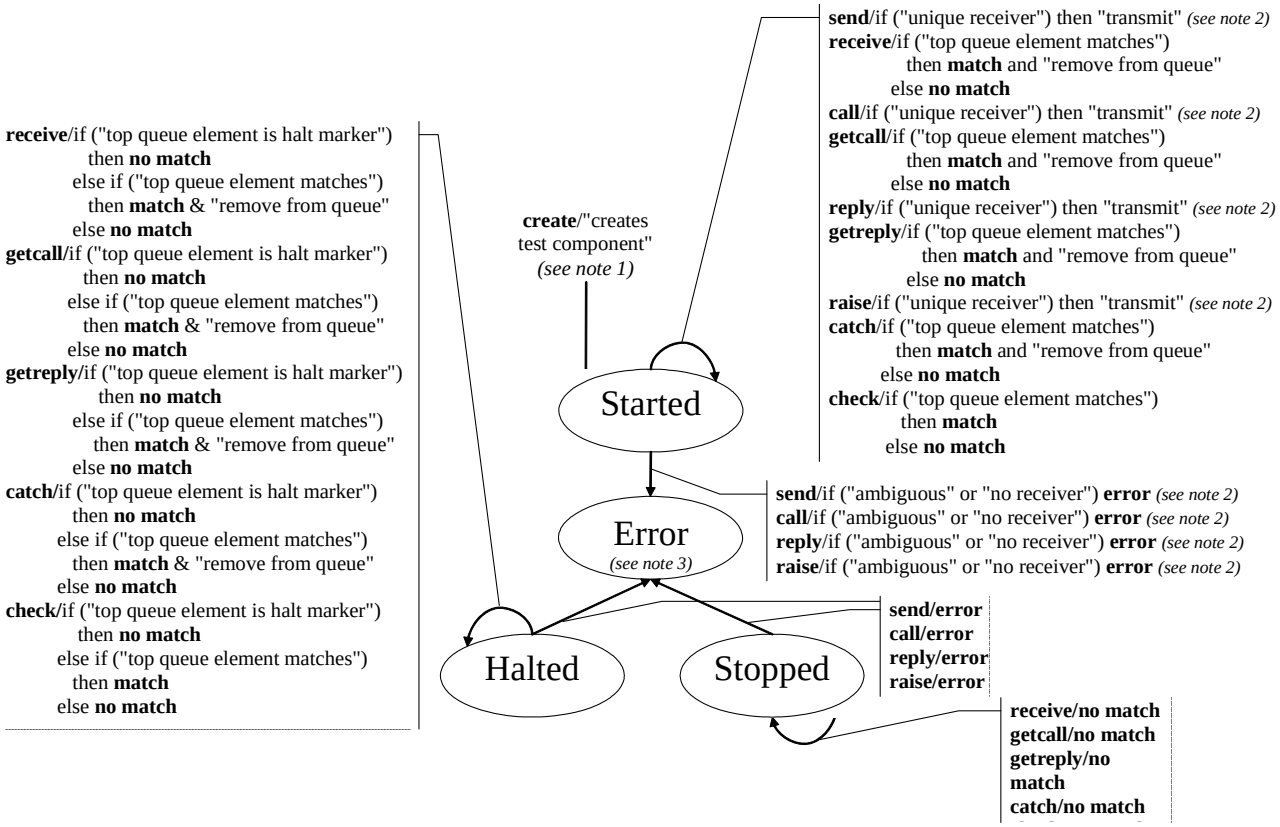


NOTE: When creating a PTC the ports of that PTC are created and started; when creating the MTC the ports of the MTC and the ports of the TSI are created and started.

Figure F.7: Dynamic behaviour of ports: port controlling operations

F.3.3 Communication Operations

The results of the communication operations send, receive, call, getcall, raise, catch, reply, getreply, check are shown in figure F.8.



NOTE 1: When creating a PTC the ports of that PTC are created and started; when creating a MTC the ports of the MTC and the ports of the TSI are created and started.
NOTE 2: A unique receiver exists if there is only one link for this port or if the to address expression references a test component whose port is linked to this port (a terminated test component is not a legal receiver).
NOTE 3: Whenever a port enters its error state, the test component it belongs to enters also its error state, assigns a local error verdict, the test case terminates and the overall test case result will be error.
NOTE 4: As trigger is a shorthand for an alt together with receive it is not considered here.

Figure F.8: Dynamic behaviour of ports: communication operations

Annex G (informative): Deprecated language features

G.1 Group style definition of module parameters

Previous versions of the present document (up to and including V2.2.1) required to use a group-like syntax shown in the example below to declare module parameters. The module parameter syntax has been unified with constant and variable declaration syntax in this version but group-like syntax is not fully removed to leave a time period for tool providers and users to change from the old syntax to the new one. The group-like syntax of module parameter declarations may be fully removed in a future edition of the standard.

EXAMPLE (superfluous syntax):

```
module MyModuleWithParameters
{
    modulepar { integer TS_Par0, TS_Par1 := 0;
                boolean TS_Par2 := true
              "};
    modulepar { hexstring TS_Par3 };
}
```

G.2 Recursive import

Previous versions of the present document (up to and including V2.2.1) allowed to import named definitions implicitly, via importing other definitions of the same module using them in a **recursive** mode. This feature is deprecated and may be fully removed in a future edition of the standard.

G.3 Using **all** in port type definitions

Previous versions of the present document (up to and including V2.2.1) allowed to use the **all** keyword in port type definitions instead of an explicit list of types and signatures allowed via the given port. This feature is deprecated and may be fully removed in a future edition of the standard.

G.4 sizeof for length of lists

Previous versions of the present document (up to and including V3.2.2) allowed to use the builtin function **sizeof** to compute the length of **record of**, **set of**, and **array**. This has been replaced by **lengthof**. The use of **sizeof** for list like types is deprecated and is planned to be fully removed in the next published edition.

G.5 sizeoftype predefined function

The previous version of the standard (up to and including V3.3.1) defined the **sizeoftype** predefined function. This feature is deprecated in this version of the standard and may be fully removed in the next published edition.

G.6 Mixed ports

Previous versions of the present document (up to and including V3.2.2) allowed to use **mixed** ports. This feature is deprecated and may be fully removed in a future edition of the standard.

G.7 External constants

Previous versions of the present document (up to and including 3.4.1) allowed to use **`external const`**ants. This feature is deprecated and may be fully removed in a future edition of the standard.

G.8 Prefixing enumerated values

Previous versions of the present document (up to and including V4.2.1) did not explicitly specify how to resolve name conflicts between imported enumeration values and global names defined in the importing or in another TTCN-3 module. Some tool implementations resolved this issue by allowing prefixing enumeration values with the name of the module in which the given enumerated type is defined. Version 4.3.1 added in clause 8.2.3.1 a rule to resolve such name clashes, therefore prefixing enumerated values is deprecated.

Annex H (informative): Bibliography

- ETSI ES 201 873-1 (V1.1.2): "Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; Part 1: TTCN-3 Core Language", 2001.
- ETSI ES 201 873-1 (V2.2.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language", 2003.
- ETSI ES 201 873-1 (V3.1.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language", 2005.
- ETSI ES 201 873-1 (V3.2.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language", 2007.
- ETSI ES 201 873-1 (V3.3.2): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language", 2008.
- ETSI ES 201 873-1 (V3.4.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language", 2008.
- ETSI ES 201 873-1 (V4.1.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language", 2009.
- ETSI ES 201 873-1 (V4.2.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language", 2010.

History

Document history		
V1.1.1	March 2001	Publication
V1.1.2	June 2001	Publication
V2.2.1	February 2003	Publication
V3.1.1	June 2005	Publication
V3.2.1	February 2007	Publication
V3.3.2	April 2008	Publication
V3.4.1	September 2008	Publication
V4.1.1	June 2009	Publication
V4.2.1	July 2010	Publication
V4.2.5	April 2011	Membership Approval Procedure MV 20110607: 2011-04-08 to 2011-06-07