ETSI ES 201 873-1 V4.9.1 (2017-05)

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

Reference
RES/MTS-201873 -1 T3ed491

Keywords
language, methodology, testing, TTCN-3

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

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 (https://ipr.etsi.org/).

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

Foreword

This ETSI Standard (ES) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS).

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)";

NOTE: Part 2 of this multi-part deliverable is in status "historical" and is not maintained.

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: "Using XML schema with TTCN-3";

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

Part 11: "Using JSON with TTCN-3";

Part 12: "Using WSDL with TTCN-3".

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

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. In addition to the textual format defined in the present document, while GFT (ETSI ES 201 873-3 [i.2]) defines a graphical presentation format for TTCN-3. 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

2.1 Normative 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 referenced 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.

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 (2014): "Information technology -- Universal Coded Character Set (UCS)".

[3] Recommendation ITU-T X.292: "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] Recommendation ITU-T T.50: "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] Recommendation ITU-T X.290: "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

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 referenced document (including any amendments) applies.

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

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] Void.

[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".

[i.15] ISO/IEC 10646 (2003): "Information technology -- Universal Coded Character Set (UCS)".

[i.16] 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)".

3 Definitions and abbreviations

3.1 Definitions

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

assignment notation: notation that can be used for record, set, record of and set of values, where the fields or the elemens to which a value is assigned are identified explicitly within a pair of curly brackets ("{" and "}") by the field names or the positions of the elements

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.

behaviour definition: Definition of dynamic test behaviour. Behaviour definitions are either `testcase`, `function`, `altstep` or module `control` part definitions.

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: Value or template is completely initialized if it is not uninitialized and, if its type is a structured type, all its required parts are completely initialized. Additionally, templates are completely initialized if they are assigned a matching mechanism all parts of which are completely initialized. If a value or template is completely initialized, it fulfills the requirement of being "at least partially initialized".

NOTE: A value or template of a simple, **`component`** or **`default`** type is completely initialized if anything but the unchanged symbol "-" has been assigned to it.
A value or template of a **`union`** or **`anytype`** type is completely initialized if one of its variants has been completely initialized.
A value or template of a **`record`** or **`set`** type with only optional fields and the **`optional "implicit omit"`** attribute attached, is completely initialized if the value "`{}`" is assigned, as all fields are implicitly set to `omit`.
A value or template of a **`record`** or **`set`** type with no fields is completely initialized with assignment of the value "{}".
A value or template of a **`record of`**, **`set of`** or array type is completely initialized if at least the first n elements are completely initialized, where n is the minimal length imposed by the type length restriction or array definition. Thus in case of n equals 0, the assignment of the value "{}" also completely initializes such a **`record of`**, **`set of`** or array.

component constant: constant defined in a component type

component port: port defined in a component type

component template: template defined in a component type

component timer: timer defined in a component type

component variable: variable defined in a component type

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

NOTE: 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

deterministic function: function that for the same input in the in and inout parameters always yields the same output both for the return result as well as the inout and out parameters

NOTE 1: A non-deterministic function is one that is not deterministic.

NOTE 2: In general, it cannot be decided if a function is deterministic or not. However, a function can be specified to be deterministic, i.e. the function is supposed to be deterministic. In this case, a violation of the determinism can be detected and handled accordingly. The handling however is tool-specific.

dynamic parameterization: form of parameterization, in which actual parameters are dependent on runtime events

EXAMPLE: The value of the actual parameter is a value received during runtime 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).

fuzzy value or template: If a value or template instance is declared to be fuzzy, the expression, initializing or partly initializing it (including actual parameters passed to in formal parameters), is subject to lazy evaluation. During execution, this expression is re-evaluated each time when the fuzzy object is referenced, except when at the left hand side of an assignment or passing it to a fuzzy or lazy formal parameters. The result of this (re)evaluation is used as the actual value or template of the fuzzy instance. When new content is assigned to a fuzzy instance or to its subpart, the right hand side of the assignment is subject to lazy evaluation again.

global visibility: attribute of an entity (module parameter, constant, template, etc.) whose 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 Recommendation ITU-T X.290 [5].

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

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

in parameterization: kind of parameterization where the value of the actual parameter (the argument) is assigned 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: In `in` parameterization, parameters are passed by value.

NOTE 2: The arguments are evaluated before the parameterized object is entered.

NOTE 3: 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.

index notation: notation to access individual elements of record of, set of, array and string values or templates, where the element to be accessed is identified explicitly by an index value enclosed in square brackets ("[" and "]") which specifies the position of that element within the referenced value or template and the index value is either an integer value, array of integers or record of integers

NOTE: Integer values used for indexing (either directly or as elements of the record of or array values) always lie within the index range of the type of the referenced value or template. Except for those arrays which are defined with an explicit index range, the index range always has 0 as the index for the first element.

initialization: value or template, or a value or template field is initialized when a content is first assigned to it

NOTE: The assignment may be explicit at the declaration of the given object, in which case the same restrictions apply as for the right-hand side of the assignment operation, or at first use on the left-hand side of an assignment, or may be implicit. Implicit initialization occurs when a yet uninitialized object is passed as actual parameter to an out formal parameter of a directly called testcase, function or altstep returns with a non-uninitialized value or template that is assigned to the actual parameter; or when module parameters not initialized in the TTCN-3 code get their runtime values before test suite execution.

inout parameterization: kind of parameterization that uses passing by reference, i.e. when the parameterized object is invoked, the formal parameter is linked with the actual parameter and gets direct access to the same data content that is currently represented by the actual parameter.

NOTE 1: The invoked object uses the actual parameter directly, so that all changes made in the formal parameter become immediately effective on the actual parameter. If the same actual parameter is passed to two distinct formal parameters, a change in one formal parameter becomes immediately effective in the other one (and in the actual parameter).

NOTE 2: Inout parameters can be used for functions, altsteps, and test cases only, if not restricted by further rules, e.g. altsteps activated as defaults.

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

lazy evaluation: Lazy evaluation means that evaluation of an expression is delayed during execution until the value or template instance, to which the result of the evaluation should have been assigned or passed to as actual parameter, is first referenced at an other place than the left hand side of an assignment or an actual parameter passed to a fuzzy or lazy formal parameter. During execution, this delayed evaluation is carried out at the first actual reference, even when the result is to be used in an expression that is also subject to lazy evaluation. For the evaluation the actual values at the time of the evaluation are to be used (not the actual values at the time of the assignment or parameter passing). This implies that components of the expression may be uninitialized at the time, when execution reaches the assignment or parameter passing, but may be initialized by the time of the evaluation that can lead to successful evaluation. If, by the time of the evaluation, execution has left the scope unit, in which one or more components of the expression is defined, the actual values of the component(s) at the time of leaving the scope unit are to be stored for the purpose of the delayed evaluation (but only for that, i.e. the values are not accessible for the user).

lazy value or template: A value or template instance is called lazy, when the expression, initializing or partly initializing it (including actual parameters passed to in formal parameters), is subject to lazy evaluation. When, during execution, the delayed (lazy) evaluation is taking place, its result is stored in the lazy value or template and the lazy instance is used further on like ordinary values and templates, until the next use of the lazy variable or parameter on the left hand side of an assignment. When a new content is assigned to a lazy instance or to its subpart, the right hand side of the assignment is subject to lazy evaluation again. If, during execution, no expression referencing the lazy object is evaluated, the lazy value or template instance is never evaluated.

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 Recommendation ITU-T X.292 [3].

out parameterization: kind of parameterization where the actual parameter's content (the argument) is not passed to the formal parameter when the parameterized object is invoked, but the content of the formal parameter is passed back to the actual parameter when the invoked object completes, if the formal parrameter has been initialized during the invocation. The actual parameter is the reference evaluated at the time of the invocation

NOTE 1: In `out` parameterization, parameters are passed by value.

NOTE 2: Out parameters can be used for functions, altsteps, and test cases only, if not restricted by further rules, e.g. **`altstep`**s activated as defaults.

NOTE 3: An **`out`** formal parameter is uninitialized (unbound) when the invoked object is entered.

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

partially initialized: value or template is partially initialized if initialization has taken place on it or to at least one of its fields or elements

NOTE: 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).

passing by reference: ability to link an actual parameter with a formal parameter of a function, altstep or test case and to control its actual value within the function, altstep or test case by using the formal parameter reference, i.e. no copy of the data content is made and the actual and formal parameters share the same data content

passing by value: ability to make a copy of a data content of an actual or formal parameter before passing it to a formal or actual parameter, i.e the actual and formal parameters do not share the same data content

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 runtime 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 Recommendation ITU-T X.290 [5].

template: TTCN-3 data objects are values or templates by definition. A TTCN-3 template identifies a subset of the values of its type (where the subset may contain a single instance of the type, several instances or all instances) or the matching mechanism **`omit`**. Templates are defined by global and local templates, template variable definitions, or formal template parameters. Any of those are templates from the point of view of their usage, irrespective of their actual content; for example, a template variable containing a specific value is a template.

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 formal template parameters are considered to be in-line templates (see clause 15.4).

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

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

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

test case error: See Recommendation ITU-T 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 Recommendation ITU-T 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

EXAMPLE: At assignments, as actual parameters at calling a function, referencing a template, etc. or as a return value of a function.

type context: "In the context of a type" means that at least one object involved in the given TTCN-3 action (an assignment, operation, parameter passing, etc.) identifies a concrete type unambiguously

NOTE: 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.).

uninitialized: value or template is uninitialized as long as no initialization of it or at least one of its parts has occurred

unqualified name: 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: TTCN-3 data objects are values or templates by definition. A TTCN-3 value is an instance of its type

NOTE: Values are defined by module parameters, constants, value variables, or formal value parameters. Any of those are value objects from the point of view of their usage. A template containing only specific value matching - though referring to a single instance of its type - is not a value object, but is a template object.

value list notation: notation that can be used for record, set, record of and set of values, where the values of the subsequent fields or elements are listed within a pair of curly brackets ("{" and "}"), without an explicit identification of the field name or element position

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
ASN	Abstract Syntax Notation
ASP	Abstract Service Primitive

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

ATS	Abstract Test Suite
BER	Basic Encoding Rules
BMP	Basic Multilingual Plane
BNF	Backus-Nauer Form
BOM	Byte Order Mark
CORBA®	Common Object Request Broker Architecture
ETS	Executable Test Suite
FIFO	First In First Out
GFT	Graphical presentation Format
ICS	Implementation Conformance Statement
IDL	Interface Definition Language
IRV	International Reference Version
ITU-T	International Telecommunication Union Telecommunication Standardization Sector
IUT	Implementation Under Test
IXIT	Implementation eXtra Information for Testing
MTC	Main Test Component
PDU	Protocol Data Unit

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

PTC	Parallel Test Component
RHS	Right Hand Side (of assignment)
SDL	Specification and Description Language
SUT	System Under Test
TCI	TTCN-3 Control Interfaces
TE	TTCN-3 Executable

NOTE: See also ETSI ES 201 873-5 [i.3].

TFT	Tabular presentation Format
TRI	TTCN-3 Runtime Interfaces
TSI	Test System Interface
TTCN-3	Testing and Test Control Notation version 3
UCS	Universal Character Set
UID	Short identifier for character code point

NOTE: See ISO/IEC 10646 [2], clauses 6.5 and 6.6.

USI	UCS Short Identifier
UTF	UCS Transformation Format
UTF-8	Unicode Transformation Format-8
UTF-16	Unicode Transformation Format-16
UTF-16BE	Unicode Transformation Format-16 big-endian
UTF-16LE	Unicode Transformation Format-16 little-endian

UTF-32	Unicode Transformation Format-32
UTF-32BE	Unicode Transformation Format-32 big-endian
UTF-32LE	Unicode Transformation Format-32 little-endian
XML	eXtensible Markup Language

4 Introduction

4.0 General

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

NOTE 1: CORBA® is the trade name of a product supplied by Object Management Group®. This information is given for the convenience of users of the present document and does not constitute an endorsement by ETSI of the product named. Equivalent products should be used if they can be shown to lead to the same results.

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 2: The present document uses the following model 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 third part, defined in ETSI ES 201 873-3 [i.2], is the graphical presentation format.

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

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

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

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

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

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

The tenth part, ETSI 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 will 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 should 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 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 ETSI 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 apply.

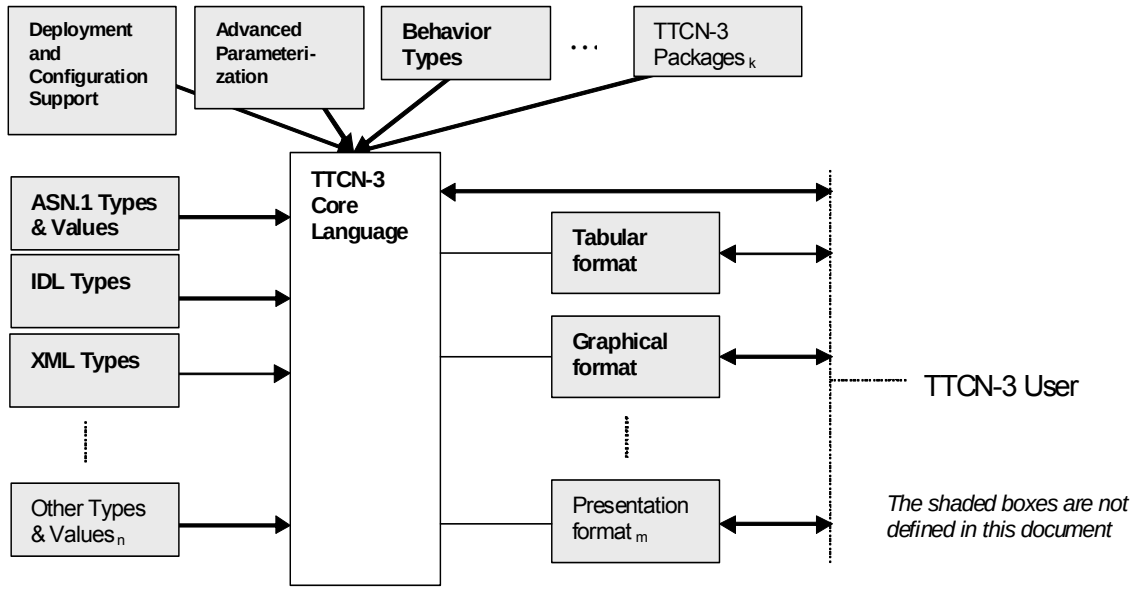


Figure 1: User's view of the core language, its packages and the various presentation formats

The core language is defined by a complete syntax (see annex A) and operational semantics (ETSI 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 ETSI ES 201 873-4 [1].

5 Basic language elements

5.0 General

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

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

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

TTCN-3 has a number of predefined 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).

Special TTCN-3 modifiers are identifiers prefixed with the @-symbol (see annex A). They modify the default semantics of the language element they are applied to in the specified way. If more than one modifier is applied to a language element, they may be applied in any order.

NOTE: These modifiers are useful for refining or modifying existing language features, for example in the context of the optional extension packages of TTCN-3 since they cannot lead to backward incompabilities with existing reserved keywords or identifiers.

5.2 Scope rules

5.2.0 General

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.

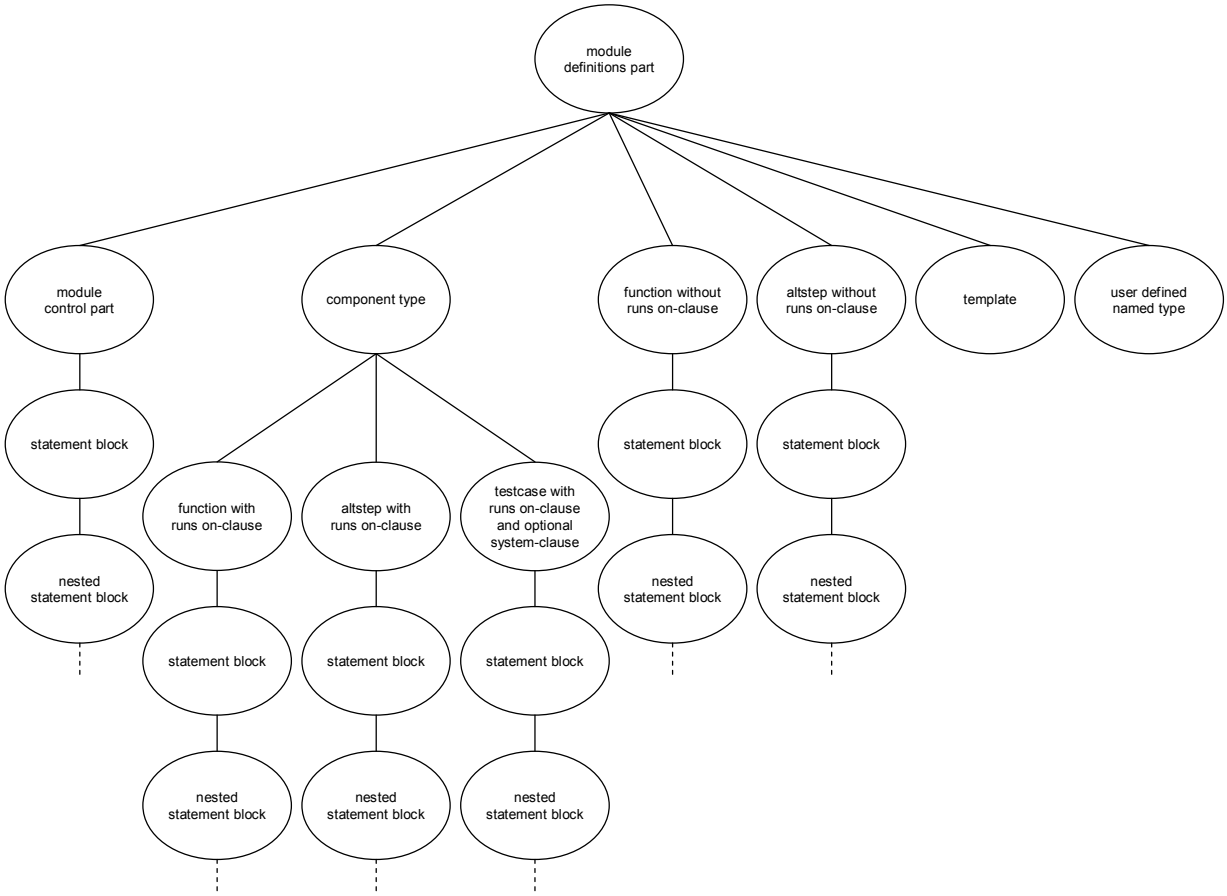


Figure 2: Hierarchy of scope units

EXAMPLE 1: Local scopes

```
module MyModule
{   :
    const integer c_myConst := 0; // c_myConst is visible to f_myBehaviourA and f_myBehaviourB
    :
    function f_myBehaviourA()
    {   :
        const integer c_a := 1;   // The constant c_a is only visible to f_myBehaviourA
        :
    }

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

    }
}
```

EXAMPLE 2: Component type scopes

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

type component MyExtendedComponentType extends MyComponentType {
    var integer vc_myVar:= 2 * cc_myConst; // using cc_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, enumerated values and groups do not have to be globally unique, however in the case of enumerated values the identifiers, within the same module, they shall only be reused for enumerated values within other enumerated types or as identifiers for fields of structured types. In addition, enumeration values shall not be used as names of value or template definitions of imported enumeration types, defining the given enumeration value (see also clause 8.2.3.1, example 4). The rules of identifier uniqueness shall also apply to identifiers of formal parameters.

EXAMPLE 1: Nested scopes

```
module MyModule
{   :
    const integer c_a := 1;
    :
    function f_myBehaviourA()
    {   :
        const integer c_a := 1; // Is NOT allowed: clash with global constant c_a
        :
        if(…)
        {   :
            const boolean c_a := true; // Is NOT allowed: clash with local constant c_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 c_a in module header)
function f_myBehaviourA()
{   :
    const integer c_a := 1;
    :
}

function f_myBehaviourB()
{   :
    const integer c_a := 1;
    :
}
```

EXAMPLE 3: Module scopes

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

    function f_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 v_myVar2 := 3;
const integer c_myConst:= 1;
if (v_myVar2+c_myConst > 10)
{
    var integer v_myVar1:= 1;
    :
    v_myVar1:= v_myVar1 + 10;
    :
}
:
```

Declarations in the module definitions part and in a component type definition may be made in any order. However inside the module control part, test case definitions, functions, altsteps, and statement blocks, all required declarations shall 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

5.4.0 General

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 runtime	all basic types, all user-defined types and **address** type.
template	Value and template parameterization	Dynamic at runtime	all basic types, all user-defined types, **address** type and **template.**
function	Value, template, port and timer parameterization	Dynamic at runtime	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 runtime	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 runtime	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

5.4.1.0 General

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.

If parameters are passed by value (i.e. in case of **`in`** and **`out`** parameters), type compatibility rules specified in clause 6.3 apply. When parameters are passed by reference, strong typing is required. Both the actual and formal parameter shall be of the same type.

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

NOTE 1: 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.

Formal value or template parameters may be declared lazy using the **`@lazy`** modifier. The behaviour of lazy parameters is defined in clause 3.1, definition of lazy values or templates. See examples in clause 5.4.1.1.

Formal value or template parameters may be declared fuzzy using the **`@fuzzy`** modifier. The behaviour of lazy parameters is defined in clause 3.1, definition of fuzzy values or templates. See examples in clause 5.4.1.1.

NOTE 2: The actual values of component variables used in the delayed evaluation of a lazy or fuzzy parameter may differ from their values at the time, when the parameterized function or alstep was called.

Assigning default values for lazy and fuzzy formal parameters does not change the parameters' semantics: when the default values are used as actual values for the parameters, they shall be evaluated the same way (i.e. delayed) as if an actual parameter was provided.

Lazy and fuzzy properties are valid only in the scope, where the parameters' names are visible. For example, if a fuzzy parameter is passed to a formal parameter declared without a modifier, it losts its fuzzy feature inside the called function. Similarly, if it is passed to a lazy formal parameter, it becomes lazy within the called function.

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 ) ] [ @lazy | @fuzzy ] 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.

NOTE 1: If functions are used for the initialization of default values of **in** parameters, it is strongly advised to avoid side effects during the evaluation of default values. Side effects may cause non-deterministic test executions.They can be avoided, e.g. by adhering to the rules defined in clause 16.1.4.

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 runtime (i.e. *static* at runtime). This means that, at runtime, 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 runtime).

NOTE 2: 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 formal parameter's default value has to be compatible with the type of the parameter. The expression may be any expression that is well-defined at the beginning of the scope of the parameterized entity, but shall not refer to other parameters of the same parameter list.

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.

j) Only in parameters can be declared lazy or fuzzy.

k) When parameters are referenced (e.g. in assignments, expressions, template bodies, etc.), the rules for variables shall apply.

Examples

EXAMPLE 1: In, out and inout formal parameters

```
function f_myFunction1(in boolean p_myReferenceParameter){ … };
// p_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 f_myFunction2(inout boolean p_myReferenceParameter){ … };
// p_myReferenceParameter is an inout value parameter. The parameter can be read and set
// within the function - the assignment is not local

function f_myFunction3(out template boolean p_myReferenceParameter){ … };
```

```
// p_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 v_f1, v_f2;
    v_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,
    v_f2 := p_int;
    return {v_f1, v_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_v := p_int1 + p_int2;
  return v_v;
}

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


type component Comp { var integer i := 0 }

function g(integer x := f_comp(i)) runs on Comp return integer {
    // reference to i from Comp is allowed in default value of parameter x
    return x;
}

function h(integer y := g()+i) runs on Comp {
    // reference to g is allowed because it has a compatible runs on clause as h
}
```

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
```

EXAMPLE 5: Lazy and fuzzy parameters

```
type component MyComp { var integer vc_int }

function f_MyLazyFuzzy(in @lazy integer p_lazy, in @fuzzy integer p_fuzzy) runs on MyComp {
```

```
  //When called from MyCalling:
  v_int := 1;
  log(p_lazy); //will log 2 as function double with actual parameter vc_int equals 1 is called
               //here; 2 is  stored in p_lazy (also, function double stores 2 in v_int)
  log(p_lazy); //will log 2 again as p_lazy is not re-evaluated
  log(p_fuzzy);//will log 4 as function double with actual parameter vc_int equals 2 is called
               // here (also, function double stores 4 in vc_int)
  log(p_fuzzy) //will log 8 as function double is re-evaluated with actual parameter 4
}

function f_double (in integer p_in) runs on MyComp return integer{
  p_in := 2* p_in;
  v_int := p_in;
  return p_in
}

testcase TC_MyCalling() runs on MyComp {
vc_int := 0;
f_myLazyFuzzy (f_double(vc_int), f_double(vc_int) )
}
```

EXAMPLE 6: Difference between passing by value and passing by reference

```
function f_byValue (in integer p_int1, in integer p_int2) {
  p_int2 := p_int2 + 1;
  log(p_int1);
  log(p_int2);
}

function f_byReference (inout integer p_int1, inout integer p_int2) {
  p_int2 := p_int2 + 1;
  log(p_int1);
  log(p_int2);
}

function f_f () {
  var integer v_int := 1;
  f_byValue(v_int, v_int); // prints 1 and 2
  log(v_int); // prints 1
  f_byReference(v_int, v_int); // prints 2 and 2
  log(v_int); // prints 2
}
```

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

Template 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. If a default template is used, it is evaluated in the scope of the parameterized entity, not the scope of the actual parameter list.

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 or altsteps 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 may be any template expression that is well-defined at the beginning of the scope of the parameterized entity, but shall not refer to other parameters in the same parameter list.

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).

h) Only in template parameters can be declared lazy or fuzzy.

i) When template parameters are referenced (e.g. in assignments, expressions, template bodies, etc.), the rules for template variables shall apply.

Examples

EXAMPLE 1: Template with template parameter

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

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

EXAMPLE 2: Function with template parameter

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

EXAMPLE 3: Template with restricted parameter

```
// The template
template MyMessageType mw_myTemplate1 (template ( omit ) integer p_myFormalParam):=
{   field1 := p_myFormalParam,
    field2 := pattern "abc*xyz",
    field3 := true
}

// could be used as follows
pco1.receive(mw_myTemplate1(omit));
// but not as follows
pco1.receive(mw_myTemplate1(?)); // AnyValue is not within the restriction
```

```
// the same template can be written shorter as
template MyMessageType mw_myTemplate2 (omit integer p_myFormalParam):=
{   field1 := p_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`** and **`altstep`** definitions may have formal timer parameters, with the exception of functions or altsteps started as test component behaviour (see clause 21.3.2).

Examples

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

// could be used as follows
function f_myBehaviour2 ()
{   :
    timer t_t;
    f_myBehaviour(t_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** and **altstep** definitions may have formal port parameters, with the exception of functions or altsteps started as test component behaviour (see clause 21.3.2).

Examples

```
// Altstep definition with a port in the formal parameter list
altstep a_myBehaviour (MyPortType p_myPort)
{   :
    [] p_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 or in a mixed notation where the first parameters are given in list notation and additional parameters in assignment notation.

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, value variables, invocations of 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 **out** formal value parameters shall be (template) variables, formal (template) parameters (of in, inout or out parameterization) or references to elements of (template) variables or formal (template) parameters of structured types. Furthermore it is allowed to use the dash symbol "-" as an actual **out** parameter, signifying that a possible result for that parameter will not be passed back.

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

NOTE 1: Reference to a string element cannot be passed by reference as string types are not structured types.

Actual parameters that are passed to **in** formal template parameters shall be literal values, module parameters, constants, variables, invocations of 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 **out** formal template parameters shall be template variables, formal template parameters or references to elements of template variables or formal template parameters of structured types. Furthermore it is allowed to use the dash symbol "-" as an actual **out** parameter, signifying that a possible result for that parameter will not be passed back.

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

When actual parameters that are passed to **in** formal value or template parameters contain a value or template reference, rules for using references on the right hand side of assignments apply. When actual parameters that are passed to **inout** and **out** formal value or template parameters contain a value or template reference, rules for using references on the left hand side of assignments apply.

The values of **out** formal parameters are passed to the actual parameters in the same order as is the order of formal parameters in the definition of the parameterized TTCN-3 object. 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.

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.

It is allowed to pass elements of structured values or templates (record, set, record of, set of, union and anytype values or templates) by reference. Modification of parameters passed this way affects the original structured value or template. Before passing the actual parameter, the rules for referencing the element on the left hand side of assignments are applied, expanding the structured value so that the referenced element becomes accessible (see clauses 6.2 and 15.6 for more details).

NOTE 2: Because inout parameters are passed by reference and component variables are effectively also accessed by reference within a called function or altstep, passing parts of a structured component variable as an actual inout parameter may have confusing effects inside the parameterized behaviour: changing either the inout parameter or the component variable may also change the other simultaneously, which might break the intended algorithm. For this reason, such situations should be avoided.

When a formal parameter is an **out** parameter or has been defined with a default value or template, respectively, then it is not necessary to provide an actual parameter. In such a case the default value or template is taken as actual parameter.

The actual parameters are evaluated in the order of their appearance. If for some formal parameters, no actual parameter has been provided, if they are **out** parameters, the dash symbol "-" and for **in** parameters their default values are taken. Default values are evaluated after the evaluation of the actual parameters and the order of their evaluation corresponds to their order in the formal parameter list.

NOTE 3: If assignment notation has been used for the actual parameter list, the order of the evaluation of actual parameters may differ from the order of the parameters in 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 **inout** parameter and for each **in** parameter without a default there shall be an actual parameter. The actual parameter of a formal **out** parameter or **in** 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 are **out** parameters or have default values and are left out. The number of actual parameters in the list notation shall not exceed the number of parameters in the formal parameter list.

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 assigned actual parameter there shall exist a corresponding formal parameter of the same name. 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) For **in** formal parameters, the type of the actual parameter shall be compatible with the type of the formal parameter. For **out** formal parameters, the type of the formal parameter shall be compatible with the type of the actual parameter. Strong typing is required for **inout** formal (parameters passed by reference). For **in** formal template parameters, the template restriction of the actual parameter shall not be less restrictive than the one of the formal parameter. For **out** formal template parameters, the template restriction of the actual parameter shall not be more restrictive than the one of the formal parameter. For **inout** formal template parameters, the template restriction of the actual and the formal parameter shall be the same.

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`**, **`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.

NOTE 4: **`signature`** objects also have formal parameters, see clauses 15.2 and 22.3 for their handling.

h) Void.

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

j) Unless specified differently in the relevant clause(s), actual parameters passed to **`in`** or **`inout`** formal parameters shall be at least partially initialized (for an exemption see e.g. clause 16.1.2 of the present document).

k) Functions, called by actual parameters passed to fuzzy or lazy formal parameters of the calling function, shall not have inout or out formal parameters. The called functions may use other functions with inout or out parameters internally.

l) Actual parameters passed to **out** and **inout** parameters shall not be references to lazy or fuzzy variables.

m) Whenever a value or template of a record, set, union, record of, set of, array and anytype type is passed as an actual parameter to an inout parameter, none of the fields or elemetns of this structured value or template shall be passed as an actual parameter to another inout parameter of the same parameterized TTCN-3 object. This restriction applies recursively to all sub-elements of the structured value or template in any level of nesting.

n) If two or more actual parameters passed to **`inout`** parameters of the same parameterized TTCN-3 object contain a reference to distinct alternatives of the same union or anytype value, an error shall be produced.

o) If the mixed notation is used, no value list notation shall be used following the first assignment notation and the parameters given in assignment notation shall not assign parameters that already have an actual parameter given in list notation.

Examples

EXAMPLE 1: Formal and actual parameter lists have to match

```
// A function definition with a formal parameter list
function f_myFunction(integer p_formalPar1, boolean p_formalPar2, bitstring p_formalPar3) { … }

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

// A function call with assignment notation for actual parameters
f_myFunction(p_formalPar1 := 123, p_formalPar3 := '1100'B, p_formalPar2 := true);
```

EXAMPLE 2: In parameters

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

// A function call with an actual parameter
f_myFunction(m_myGlobalTemplate);
```

EXAMPLE 3: Inout and out parameters

```
function f_myFunction(inout boolean p_myReferenceParameter){ … };
// p_myReferenceParameter is an inout parameter

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


function f_myFunction(out template boolean p_myReferenceParameter){ … };
```

```
// p_myReferenceParameter is an out parameter

// A function call with an actual parameter
f_myFunction(v_myBooleanVariable);
// The actual parameter is initially unbound, but can be set and read within the function.
f_myFunction(-); // the outcoming value is not assigned to a variable

type record of integer RoI;

function f_swapElements (inout integer p_int1, inout integer p_int2) {
  var integer v_tmp := p_int1;
  p_int1 := p_int2;
  p_int2 := v_tmp;
}

function f_testReferences (inout RoI p_roi, inout integer p_elem) { … }
:
var RoI v_roi := { 0, 1, 2, 3, 4, 5 };
f_swapElements(v_roi[0], v_roi[5]); // after the function call, v_roi is { 5, 1, 2, 3, 4, 0 }
f_testReferences(v_roi, v_roi[2]); // produces an error as elements of v_roi are not allowed
    // to be passed by reference if the parent structure (v_roi) is passed by reference too.

function f_changeAndIncrement(out integer p_e, in integer p_v, inout integer p_i) {
    p_i := p_i + 1;
    p_e := p_v;
}
:
var integer v_i := 0;
f_changeAndIncrement(v_roi[v_i], 3, v_i); // increments p_i, but still assigns 3 to v_roi[0]
```

EXAMPLE 4: A side effect caused by passing part of a component variable as inout parameter

```
type component MyComp {
    var ROI v_rec := { 0, 1 }
}

testcase TC() runs on MyComp {
    f_test(v_rec[1]) // passing 2nd element of component variable as inout parameter
    log(v_rec); //will log { 2 , 2 }
}

function f_test(inout integer p_int) runs on MyComp {
    v_rec := { 2 }; // now, isbound(p_int) == false
    p_int := 2; // now, v_rec == { 2, 2 }
}
```

EXAMPLE 5: Empty parameter lists

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

// and shall be called as
f_myFunction();


// A template definition with a default value for a formal parameter written as
template MyRecord m_mytemplate (integer p_myValue:= 1):= { … }

// may be used without actual parameter list  (i.e. the default value is used)
myPCO.send(m_mytemplate)
```

EXAMPLE 6: Nested parameter lists

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

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

```
// A test case parameterized with a template might be
testcase TC_001(template MyMessageType p_rxMsg) runs on PTC1 system TS1 {
    :
    myPCO.receive(p_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 shall be provided
control
{   :
    execute(TC_001(mw_myTemplate(7)));
    :
}
```

EXAMPLE 7: A typical use case for lazy parameterization

```
modulepar boolean PX_LOG_MESSAGE := true;

function f_logMsg(@lazy charstring p_complex) {
  if (PX_LOG_MESSAGE) {
    log(p_complex);
  }
}

function f_computeComplexMessage() return charstring {
    // some complicated computation
}

f_logMsg(f_computeComplexMessage()); // f_computeComplexMessage() is only invoked if
                                     // PX_LOG_MESSAGE is true
```

EXAMPLE 8: Actual parameters passed to lazy and fuzzy formal parameters

```
type record MyMessage { integer id, float number }

type port MyPortType message { inout MyMessage }

type component MyMTC {
  var integer vc_id;
  port MyPortType p;
}

testcase TC_shootingMessages () runs on MyMTC {
  connect(self:p,self:p);
  f_sendLazy({vc_id, rnd()}); //note that at this point vc_id is unintialized yet
  f_sendFuzzy({vc_id, rnd()})
}

function f_sendLazy(@lazy MyMessage p_pdu) runs on MyMTC {
  for (vc_id := 1; vc_id<9; vc_id:=vc_id+1){
  p.send(p_pdu); // the actual parameter passed to the formal parameter p_pdu is evaluated only
                 // in the first loop;let say rnd() returns 0.924946;
                 // the message { 1, 0.924946 } is sent out 8 times
  }
  setverdict(pass,"messages has been sent out")
}

function f_sendFuzzy(@fuzzy MyMessage p_pdu) runs on MyMTC {
  for (vc_id := 1; vc_id<9; vc_id:=vc_id+1){
  p.send(pdu); // the actual parameter passed to the formal parameter p_pdu is evaluated in each
               // loop; let say rnd() returns 0.924946, 0.680497, 0.630836, 0.648681, 0.428501,
               // 0.262539, 0.646990, 0.265262 in subsuent calls; the messages  1, 0.924946 },
               // {{ 2, 0.680497 }, { 3, 0.630836 }, { 4, 0.648681 }, { 5, 0.428501 },
               //  { 6, 0.262539 }, { 7, 0.646990 } and { 8, 0.265262 } are sent out in sequence
  }
  setverdict(pass,"messages has been sent out")
}
```

EXAMPLE 9: Order of out parameters

```
function f_initValues (out integer p_par1, out integer p_par2) {
  p_par1 := 1;
  p_par2 := 2;
}

function f_f(){
    var integer v_var1;
```

```
    f_initValues(p_par2 := v_var1, p_par1 := v_var1);
    // After this function call, v_var1 will contain 2, as parameters are assigned in
    // the same order as in the definition of the f_initValues function. Thus p_par1 is
    // assigned first to v_var1 and p_par2 after that overwriting the previous value.
}
```

EXAMPLE 10: Skipped actual parameters

```
function f_skip (out integer p_par1, in integer p_par2 := 2) {
  p_par1 := 1 + p_par2;
}

function f_f(){
    // the following statements all have the same semantics :
    f_skip (-,-); // p_par2 is initialized with default value 2 and
                  // the result of p_par1 is not assigned to any variable
    f_skip (p_par1 := -, p_par2 := -);
    f_skip (p_par2 := -); // skip p_par1
    f_skip (-) ;  // skip p_par2 because it is the last
    f_skip () ;   // skip p_par1 because all following are also skipped
}
```

EXAMPLE 11: Mixed notation

```
function f_mixed (out integer p_par1, in integer p_par2 := 2, inout integer p_par3) {
  p_par1 := 1 + p_par2;
}

function f_f(){
    var integer v := 0;
    // the following statements all have the same semantics:
    f_mixed(-,2,v);
    f_mixed(-,p_par2 := 2, p_par3 := v);
    f_mixed(-,-,p_par3 := v);
    f_mixed(-,p_par3 := v, p_par2 := 2);

    // not allowed:
    f_mixed(-,2,p_par3 := v, p_par2 := 5); // p_par2 is already assigned in list notation
}
```

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 c_cConst := { 1 , c_dConst.b}; // c_cConst refers to c_dConst
    const ARecordType c_dConst := { 1 , c_cConst.b}; // c_dConst refers to c_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

6.0 General

TTCN-3 supports a number of predefined basic types. These basic types include ones normally associated with a programming language, such as **integer**, **boolean** and string types, as well as some TTCN-3 specific ones such as **verdicttype**. Structured types such as **record** types, **set** types and **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 defined within a TTCN-3 module. In any specific module context, only the known types can be accessed in a value or template of type **anytype**.

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

6.1.1.0 General

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.

Within the quotes any number of whitespaces or any sequence of the following C0 control characters: LF(10), VT(11), FF(12), CR(13) which constitutes a newline (see Recommendation ITU-T T.50 [4]) (jointly called newline characters, see clause A.1.5.1) may be included. The newline shall be preceded by a backslash ("\"). Such whitespaces, control characters and backslash will be ignored for the value and length calculation.

EXAMPLE 1:
```
'01101'B
'0110 1001'B
'0110\
 1001'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.

Within the quotes any number of whitespaces or any sequence of the following C0 control characters: LF(10), VT(11), FF(12), CR(13) which constitutes a newline (see Recommendation ITU-T T.50 [4]) (jointly called newline characters, see clause A.1.5.1) may be included. The newline shall be preceded by a backslash ("\"). Such whitespaces, control characters and backslash will be ignored for the value and length calculation.

EXAMPLE 2:
```
'AB01D'H
'ab01d'H
'Ab01D'H
'Ab 01 D'H
'Ab\
 01\
 D'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.

Within the quotes any number of whitespaces or any sequence of the following C0 control characters: LF(10), VT(11), FF(12), CR(13) which constitutes a newline (see Recommendation ITU-T T.50 [4]) (jointly called newline characters, see clause A.1.5.1) may be included. The newline shall be preceded by a backslash ("\"). Such whitespaces, control characters and backslash will be ignored for the value and length calculation.

EXAMPLE 3:
```
'FF96'O
'ff96'O
'Ff96'O
'Ff 96'O
'Ff\
 96'O
```

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

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

```
const charstring c_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.1.2) with the positive integer value of their encoding as argument, by a "quadruple" or using the USI-like notation.

NOTE 4: If applying the double quote format all characters from any character set defined in ISO/IEC 10646 [2] are allowed. Users should be aware of the character set capabilities of their editing tool and the TTCN-3 module transfer syntax UTF-8 (see clause 8).

NOTE 5: 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 Latin small letter u with double acute: "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 expression: "the Braille character" & **char** (0, 0, 40, 48) & "looks like this" represents the literal string: the Braille character ⣿ looks like this.

The UCS sequence identifier-like (USI-like) notation (see also clause 6.6 of ISO/IEC 10646 [2]) can be used to denote 1..N characters, using their short identifiers of code point (similar to UIDs described in clause 6.5 of ISO/IEC 10646 [2]). The USI-like notation is composed of the keyword **char** followed by parentheses. The parentheses enclose a comma-separated list of short identifiers . Each short identifier represents a single character and it shall be composed of a letter **U** or **u** followed by an optional "+" PLUS SIGN character, followed by 1..8 hexadecimal digits. The hexadecimal digits represent the numeric code point of the character. (e.g. `char(U0171)` denotes the Latin small letter u with double acute: "ű"). In the USI-like notation, the leading zeroes can be omitted, (i.e. `char(U171)` is equal to `char(U0171))`.

EXAMPLE 6: The expression: **char** (U4E2D, U56FD) represents the literal string: 中国.

NOTE 6: Control characters can be denoted by using the predefined conversion function, the quadruple form or the USI-like notation.

By default, **universal charstring** shall conform to the UTF-32 encoding specified in clause 9.3 of ISO/IEC 10646 [2].

NOTE 7: UTF-32 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 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.

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

For accessing individual string elements the following rules apply:

- Only single elements of the string may be accessed. Trying to assign strings with length 0 or more than 1 to a string element using the array-like syntax shall cause an error.
- 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. Trying to retrieve an element from a string with an index outside this range shall cause an error.
- For assigning an element to the end of a string, the length of the string should be used as index. Trying to assign an element to the end of a string with an index larger than the length of the string shall cause an error.
- For initializing an uninitialized string with a single element, the index value zero (0) can be used as index. Trying to assign a single element to an uninitialized string with an index which is not zero (0) shall cause an error.

EXAMPLE 1: Accessing an existing element

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

EXAMPLE 2: Specific cases

```
var bitstring v_myBitStringA, v_myBitStringB, v_myBitStringC, v_myBitStringD;
v_myBitStringA := '010'B;
v_myBitStringA[1] := '11'B; //causes an error as only individual elements can be accessed

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

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

// v_myBitStringD is not initialized
v_myBitStringD[0] := '0'B; // value of v_myBitStringD is '0'B

v_myBitStringD[1] := '1'B; // value of v_myBitStringD is '01'B

var charstring v_myCharString;
v_myCharString[0] := "a" //initializing v_myCharString with a single character
v_myCharString[1] := "" //causes an error as the length of the to-be-assigned string is 0
v_myCharString[1] := "bc" //causes an error as the length of the to-be-assigned string is
                          //more than 1
```

6.1.2 Subtyping of basic types

6.1.2.0 General

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 templates

TTCN-3 permits the specification of a list of distinguished templates as listed in table 3. The templates in the list shall be instances of the type being constrained and the set of values matching at least one of these templates 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 matching at least one of the templates in the list. The templates in the list shall only (directly or indirectly) reference other templates or constant expressions. Constant expressions used (directly or indirectly) in the template expressions shall meet with the restrictions in clause 10 for constant expressions used in type definitions.

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 (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 integer MyIntegerRange (-infinity .. -1);  // all negative integer numbers

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** should 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 sub typed. 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 charstring MyStringCaseAgnostic (pattern @nocase "abc*xyz");
    // all permitted values of MyStringCaseAgnostic have a
    // prefix abc or Abc or aBc or abC or ABc or aBC or AbC or ABC, and a
    // postfix xyz or Xyz or xYz or xyZ or XYz or xYZ or XyZ or 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 template 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, template 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 from 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 from 3 to 9 characters containing characters
    // a to z

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

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

6.2.0 General

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

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

EXAMPLE 1:

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

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

The assignment notation can be used for **record, record of**, **set**, **set of** and **union** value notations and for arrays. In this notation each field shall not appear more than once. The value list notation can be used for **record, record of**, **set** and **set of** value notations and for arrays. The index notation can be used for **record of** and **set of** value notations and for arrays. In this notation each index shall not appear more than once and shall conform to the range of indices allowed by the type definition. See more details in the subsequent clauses.

EXAMPLE 2:

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

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

// or
var MyRecordType v_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 c_myRecordValue:= {c_myIntegerValue, field2 := true, "A string"}
```

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 4:

```
// 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 5:

```
// 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

6.2.1.0 General

TTCN-3 supports ordered structured types known as **record**. The fields 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 field 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 field 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 v_myIntegerValue := 1;

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

var MyRecordType v_myRecordValue :=
{
    field1 := v_myIntegerValue,
    field2 := c_myOtherRecordValue,
    field3 := "A string"
}
```

The same value specified with a value list.

EXAMPLE 4:

```
v_myRecordValue:= {v_myIntegerValue, {'11001'B, true}, "A string"};
```

When the assignment notation is used for **record**-s, fields wished to be changed shall be identified explicitly and a value, the not used symbol "-" or the **omit** keyword can be associated with them. The **omit** keyword shall only be used for optional fields. Its result is that the given field is not present in the given value. Mandatory fields, not explicitly referred to in the notation or explicitely unspecified using the not used symbol "-", shall remain unchanged. In particular, when specifying partial values (i.e. setting the value of only a subset of the fields) using the assignment notation, at initialization, only the fields to be assigned values shall be specified. Fields not mentioned are implicitly left uninitialized. When re-assigning a previously initialized value, using the not used symbol or just skipping a field in an assignment notation, will cause that field to remain unchanged. Even when specifying partial values each field shall not appear more than once.

NOTE: Please note the difference between omitted and uninitialized fields. Omitted optional fields are not present in the record or set value intentionally, i.e. the field is initialized and it does not prevent the whole record or set from being completely initialized.

EXAMPLE 5:

```
type record MyRecordType
{
    bitstring       field1,
    boolean         field2 optional,
    charstring      field3
}

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

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

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

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

When the assignment notation is used in a scope, where the **optional** attribute is implicitly or explicitly set to "explicit omit", optional and mandatory fields, not directly referred to in the notation shall remain unchanged. When optional fields of variables are not assigned explicitly, they are uninitialized (i.e. the optional attribute shall not have any effect on variables as described in clause 27.7 restriction a)).

When the assignment notation is used in a scope, where the **optional** attribute is set to **"implicit omit"**, optional fields, not directly referred to in the notation, shall implicitly be set to omit, while mandatory fields shall remain unchanged (see also clause 27.7).

EXAMPLE 6:

```
type record MyRecordType
{
    bitstring       field1,
    boolean         field2 optional,
    charstring      field3
}

const MyRecordType c_myConst1 :=
{
    field1 := '111'B,
    field3 := “A string”
} //    { '10111'B, <undefined>, “A string”}

const MyRecordType c_myConst2 :=
{
    field1 := '111'B,
    field3 := “A string”
} with { optional "implicit omit" }
//  { '10111'B, omit /* because of the optional attribute */, “A string”}
```

When using the value list notation, all fields listed in the notation shall be specified either with a value, the not used symbol "-" or the **omit** keyword. The **omit** keyword shall only be used for optional fields. Its result is that the given field is not present in the given value. The first component of the list (a value, a "-" or **omit**) is associated with the first field, the second list component is associated with the second field, etc. No empty assignment is allowed (i.e. two commas, the second immediately following the first or only with white space between them). Fields to be left unchanged, but followed by fields to which a value or template is assigned explicitly, shall be skipped by using the not used symbol "-".

When using value list notation in a scope where the **optional** attribute is implicitly or explicitly set to "explicit omit", all remaining fields at the end of the type definition, missing from the value list notation,are left unchanged.

When using value list notation in a scope where the **optional** attribute is set to **"implicit omit"**, optional fields wished to be omitted by the implicit mechanism, but followed by fields to which a value or template is assigned explicitly, shall be skipped by using the not used symbol "-". When all remaining fields at the end of the type definition are optional and they are wished to be omitted by the implicit mechanism, either the not used symbol "-" can be used for some or all of them or they can simply be left out from the notation.

EXAMPLE 7:

```
type record R {
    integer f1,
    integer f2 optional,
    integer f3,
    integer f4 optional,
    integer f5 optional
}

const R c_x := { 1, -, 2 } with { optional "implicit omit" }
// after the assignment v_x contains { 1, omit, 2, omit, omit }
constR c_x2 := { 1, 2, 3, - } with { optional "implicit omit" }
//  after the assignment v_x2 contains { 1, 2, 3, omit, omit }
```

When using direct assignment notation in a scope where the **optional** attribute is set to "implicit omit", the uninitialized optional fields in the referenced value, shall implicitly be set to omit after the assignment in the new value, while mandatory fields shall remain unchanged (see also clause 27.7).

EXAMPLE 8:

```
const R c_x3 := { 1, -, 2 }
// after the assignment c_x3 contains { 1, <undefined>, 2, <undefined>, <undefined>}
const R c_x4 := c_x3 with { optional "implicit omit" }
//  after the assignment c_x4 contains { 1, omit, 2, omit, omit }
```

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:

```
v_myVar1 := v_myRecord1.myElement1;
// If a record is nested within another type then the reference may look like this
v_myVar2 := v_myRecord1.myElement1.myElement2;
```

EXAMPLE 2:

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

If a field in a **record** type or a subtype of a **record** type is referenced by the dot notation, the resulting type is the set of values allowed for that field imposed by the constraints of the field declaration itself (i.e. any constraints applied to the **record** type itself are ignored).

EXAMPLE 3:

```
type record MyType2
{
    integer field1 (1 .. 10),
    charstring  field2  optional
}

type MyType2 MyType3 ({1, omit}, {2, "foo"}, {3, "bar"}) ;

type MyType3.field1 MyType4;    // MyType4 is the integer type constrained to
                                // the values 1..10
type MyType3.field2 MyType5;    // MyType5 is the charstring type
type MyType2.field1 MyType6;    // MyType6 is the integer type constrained to
                                // the values 1..10
type MyType2.field2 MyType7;    // MyType7 is  the charstring type
```

Referencing a subfield of an uninitialized or omitted record field or value on the right hand side of an assignment shall cause an error.

EXAMPLE 4:

```
type record MyType4
{
    integer field1 optional,
    record
    {
        integer subfield1,
        integer subfield2
    } field2 optional
}
...
var MyType4 v_rec := { field1 := 1, field2 := omit }
var integer v_int := v_rec.field2.subfield1;
    // causes an error as v_rec.field2 is omitted
```

When referencing a field of an uninitialized record value or field or omitted field (including omitting a field at a higher level of the embedding hierarchy) on the left hand side of an assignment, the reference shall recursively be expanded up to and including the depth of the referenced subfield as follows:

a) When expanding a value or value field of record type, the subfield referenced in the dot notation shall be set to present and all unreferenced mandatory subfields shall be left uninitialized; when the assignment is used in a scope where the **`optional`** attribute is equal to `"explicit omit",` all unreferenced optional subfields shall be left undefined. When the assignment is used in a scope where the optional attribute is equal to `"implicit omit"`, all unreferenced optional subfields shall be set to **`omit`**.

b) Expansion of **`record of`**/**`set of`**/**`array`**, **`union`** and **`set`** values and intermediate fields shall follow the rules of item a) in clauses 6.2.3 and 6.2.5.1 and clause 6.2.2.1 correspondingly.

At the end of the expansion, the value at the right hand side of the assignment shall be assigned to the referenced subfield.

EXAMPLE 5:

```
var MyType4 v_rec;
v_rec.field2.subfield1 := 5;
    // after the assignment v_rec is { field1 := <undefined>, field2 := { subfield1 := 5,
    // subfield2 := <undefined> } }
```

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:

```
v_myRecordValue:= {v_myIntegerValue, omit , "A string"};

// Note that this is not the same as writing,
// v_myRecordValue:= {v_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

6.2.2.0 General

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

EXAMPLE:

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

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

NOTE: When the value list notation is used for values of **set** types, the values are assigned to the fields in the sequential order of the fields in the type definition.

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. For referencing field types of **set** types, the same rules apply as in clause 6.2.1.1 for fields of **record** types.

EXAMPLE:

```
v_myVar3 := v_mySet1.myElement1;
// If a set is nested in another type then the reference may look like this
v_myVar4 := v_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

6.2.3.0 General

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

NOTE 1: For the 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
```

When the assignment notation is used for **record of**-s and **set of**-s, elements wished to be changed are identified explicitly and either a value or the not used symbol "-" can be assigned to them. Other elements, not referred to in the notation, shall remain unchanged. In particular, when specifying partial values (i.e. setting the value of only a subset of the fields) using the assignment notation, for example, at initialization, only the elements to be assigned values shall be specified: elements not mentioned are implicitly left uninitialized. It is also possible to leave fields explicitly unspecified using the not used symbol "-". When re-assigning a previously initialized value, using the not used symbol or just skipping a field or element in an assignment notation, will cause that field or element to remain unchanged.

EXAMPLE 2:

```
var MyRecordOfType v_myVariable := {
    [0] := '111'B,
    [1] := '101'B,
    [2] := -
}

v_myVariable := { '10111'B, -, - };
// after this, v_myVariable contains:
//  { '10111'B, '101'B /* unchanged */, <undefined> /* unchanged */ }

v_myVariable :=
{
    [1] := '010'B,
}
// after this, v_myVariable contains:
//  { '10111'B/* unchanged */, '010'B, <undefined>/* unchanged */ }

v_myVariable :=
{
    [0] := -,
    [1] := '001'B,
    [2] := -
}
// after this, v_myVariable contains:
//  { '10111'B/* unchanged */, '001'B, <undefined> /* unchanged */}
```

When using the value list notation, all elements in the structure shall be specified either with a value or the not used symbol "-". The first member of the list is assigned to the first element, the second list member 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 "-". Already initialized elements left without a corresponding list member in a value list notation (i.e. at the end of a list) are becoming uninitialized. In this way, a value with initialized elements can be made empty by using the empty value list notation ("{}").

Index notation can be used on both the right-hand side and left-hand side of assignments. The index notation, when used on the right-hand side, refers to the value of the identified element of a **record of** or a **set of**. When it is used at the left-hand side, only the value of the identified single element is changed, values assigned to other elements already remain unchanged. 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 (uninitialized), this shall cause a semantic or runtime error. Referencing an identified element of an uninitialized or omitted record of or set of field or value on the right-hand side of an assignment shall cause an 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 uninitialized value.

For nested record of or set of types, an array or record of integer restricted to a single size can be used as a short-hand notation for a nested index notation.

When referencing an element of an uninitialized record of or set of value or field or omitted field (including omitting a field at a higher level of the embedding hierarchy) on the left-hand side of an assignment, the reference shall recursively be expanded up to and including the depth of the referenced element as follows:

a) When expanding a value or value field of **`record of`** or **`set of`** type, the element referenced by the index notation shall be set to present and all elements with a smaller index shall be created with an uninitialized value.

b) Expansion of **`record`**, **`union`** and **`set`** values and intermediate fields shall follow the rules of item a) in clauses 6.2.1.1 and 6.2.5.1 and clause 6.2.2.1 correspondingly.

c) At the end of the expansion, the value at the right-hand side of the assignment shall be assigned to the referenced element.

Uninitialized elements are permitted only in transient states (while the value remains invisible). Sending a **`record of`** or **`set of`** value with uninitialized elements shall cause an error.

NOTE 2: When using on the right-hand side of an assignment for **`record of-`**s or **`set of-`**s, the assignment notation and the indexed notation have similar effect, with the exception that the assignment notation is able to address multiple elements in one notation, while the index notation is able to address a single element only.

EXAMPLE 3:

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

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

// Using an index notation
v_myVar := v_myRecordOfVar[0]; // the first element of the "record of" value (integer 0)
                               // is assigned to v_myVar

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

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

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

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

// Expansion of uninitialized record of value:
v_recof[1][2] := 0;
    // after the assignment v_recof is { <undefined>, { <undefined>, <undefined>, 0 } }

// Pls. Note the difference between the two index assignment notations in
// the following example:
var MyRecordOf v_ix := { 0,1,2 }
v_ix := { [3] := 2*v_ix[2]+1 }
// the value of v_ix is: { 0, 1, 2, 5 }

// The same result can be achieved by using an index notation on the left hand side of
// the assignment:
var MyRecordOf v_ix := { 0,1,2 }
v_ix[3] := 2*v_ix[2]+1
// the value of v_ix is: { 0, 1, 2, 5 }
```

NOTE 3: 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 p_src) return MyRecordOf
{
var MyRecordOf v_dest;
var integer v_i, v_srcLength := lengthof (p_src);
for(v_i := 0; v_i < v_srcLength; v_i := v_i + 1) {
    v_dest[v_srcLength - 1 - v_i] := p_src[v_i];
}
return v_dest;
}
```

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

EXAMPLE 4:

```
// 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 v_myRecordOfArray;
// and reference to a particular element would look like this
// (value of the second element of the third 'MyBasicRecordOfType' construct)
v_myRecordOfArray [2][1] := 3;

//with the short-hand notation this could also have been written as
var integer v_i[2] := { 2, 1 };
v_myRecordOfArray [v_i] := 3;
// is the same as assigning element v_myRecordOfArray[v_i[0]][v_i[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`*. If the type definition restricts the element type of the **record of** or **set of** type, referencing the inner type of that type yields a type which contains all values from the constrained type.

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 ;

type record length (5) of record of integer ConstrainedRecordOfInt (1 .. 10);
type ConstrainedRecordOfInt[-] ConstrainedInt;
// defines the type record of integer, where the integer values are restricted
// to the range 1 .. 10 but the record of has no length restriction
```

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 enumerated values. Each enumerated value shall have an identifier and referencing the values shall only use these identifiers. The identifiers of enumerated values 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. This means that for any instantiation or value reference of an **enumerated** type, the given type shall be implicitly or explicitly identified.

NOTE 1: For example, if the enumerated type is an element or field of a user defined structured type, the enumerated type is implicitly referenced via the given element/field (i.e. by the identifier of the field or the position of the value in a value list notation) at value assignment, instantiation, etc. Another example is passing an enumerated value as actual parameter, in which case the type of the corresponding formal parameter establishes the type context needed to make the enumeration value visible. The third example is the comparision operators: if the type of one of the operands is uniquely identified, it is used as a type context for the other operand (see example 2 below). The fourth example is the match operation, where the type of the template parameter establishes the type context for the operation, if the type of the value parameter is not identified (see example 2 in clause 15.9).

The identifiers of enumerated values, within the same module, 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).

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 identifier within
// a different enumerated type

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

type record MyNewRecordType {
    MyFirstEnumType firstField,
    integer         secondField
};

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

EXAMPLE 2: Using enumerated types (see also example 4 of clause 8.2.3.1)

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

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

// The following operation is correct
if (v_today == Monday ) {...}
// the type of variable v_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
```

Each enumerated value may optionally have a user-assigned integer value or non-empty list of integer literal values or ranges of integer literal values, which is defined after the name of the enumerated value in parenthesis. Each user-assigned integer number shall be distinct within a single **`enumerated`** type, all ranges of all the values lists shall be disjoint and shall not include any of the used single integer values. For each enumerated value without an assigned integer value, the system successively associates an integer number in the textual order of the enumerated values, starting at the left-hand side, beginning with zero, by step 1 and skipping any number occupied by any of the enumerated values with a manually assigned value or value list. These values are only used by the system to allow the use of relational operators. Enumerated names with an associated value list shall only be used as values together with a specific integer value, which shall be one from the associated list, in parenthesis after the name. They can be used as a template of the enumerated type by adding a list of integer template(s) in parenthesis after the name. For enumerated values with no value assigned or with a specific integer value assigned, the user shall not directly use associated integer values but can access them and convert integer values into enumerated values by using the predefined functions **`enum2int`** and **`int2enum`** (see clauses 16.1.2, C.1.30 and C.1.4).

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).

EXAMPLE 3: Enumeration example with associated integers

```
type enumerated MyThirdEnumType {
    Blue(0),
    Yellow(1),
    Green(3),
    Other(2, 4..255)
}

var MyThirdEnumType v_color := Other(5);
if (v_color == Other(4)) { // is false
}
if (v_color > Other(4)) { // is true
}
if (match(v_color, Other(?))) { // is true
}
if (match(v_color, Other(6..10))) { // is false
}
v_color := Blue(0) //causes an error as enumerated values with a specific integer value assigned
                   //shall not use the associated integer value
```

When a TTCN-3 module parameter, formal parameter, constant, variable, non-parameterized template or parameterized template with all formal parameters having default values of an imported enumerated type is defined, the name of that definition shall not be the same as any of the enumerated values of that type.

6.2.5 Unions

6.2.5.0 General

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 1:

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

// A valid instantiation of MyUnionType would be
var MyUnionType v_age, v_oneYearOlder;
var integer v_ageInMonths;

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

```
v_ageInMonths := v_age.number * 12;
```

The assignment notation shall be used for **union**-s, and the notation shall assign a value to one field only. This field becomes the chosen field. Neither the not used symbol "-" nor **omit** is allowed in union value notations.

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

At most one of the union alternatives can be declared as the default alternative by using the **@default** modifier before the type of the alternative. For unions with a default alternative, special type compatibility rules apply (see clause 6.3.2.4) which allow using the union value as compatible with the type of the default alternative. Therefore, the assignment notation does not have to be used to denote a value of the union type if the union's default alternative is to be chosen. Also, the default alternative selection does not have to be used to access the default alternative, if it is chosen.

EXAMPLE 2:

```
type union MyUnionTypeWithDefault
{
  @default integer          number,
           charstring       string
};

// A valid instantiation of MyUnionTypeWithDefault would be
var MyUnionTypeWithDefault v_age, v_oneYearOlder;

v_age := 34;    // implicit usage of the default alternative: the integer type is
                // compatible with the default alternative; this is a shorthand notation
                // for v_age.number := 34 or v_age := { number := 34 }

v_oneYearOlder := v_age+1;  // implicit selection of the default alternative: the union
                            // default alternative is compatible with integer, so that it
                            // can be used as an integer expression; this is equivalient to:
                            // v_oneYearOlder.number := v_age.number+1;

type union MyDefaultUnionType2 {
    @default
    MyDefaultUnionType ageInYears,
    integer ageInDays
}

var MyDefaultUnionType2 v_age2 := 3;    // nested default usage: 3 is compatible with
                                        // both alternatives, but only alternative ageInYears
                                        // has @default, so this is equivalent to
                                        // v_age2 := { ageInYears := 3 } which is equivalent

                                        // to v_age2 := { ageInYears := { number := 3 } }
var integer v_result := v_age + v_age2; // v_result is 37 as the expression is equivalent
                                            // to v_age.number + v_age2.ageInYears
v_age := {string := "I feel young"};
v_result := v_age + v_age2;         // test case error: v_age would be treated as
                                        // v_age.number, which is not the selected alternative
```

6.2.5.1 Referencing fields of a union type

Alternatives of a **union** type shall be referenced by the dot notation *TypeIdOrExpression.AlternativeId*, where *TypeIdOrExpression* resolves to the name of a union type or an expression of a union type such as variable, formal parameter, module parameter, constant, template, or function invocation. *AlternativeId* shall resolve to the name of an alternative in the union type or in case of an anytype value or template *AlternativeId* shall resolve to a known type name or a known type name qualified with a module name. Alternatives of union type definitions shall not reference themselves.

EXAMPLE 1:

```
v_myVar5 := v_myUnion1.myChoice1;
// If a union type is nested in another type then the reference may look like this
v_myVar6 := v_myRecord1.myElement1.myChoice2;
// Note, that the union type, of which the field with the identifier 'myChoice2' is referenced,
// is embedded in a record type
```

If an alternative in a union type or a subtype of a union type is referenced by the dot notation, the resulting type is the set of values allowed for that alternative imposed by the constraints of the alternative declaration itself (i.e. any constraints applied to the union type itself are ignored).

When an alternative of a union type is referenced on the right hand side of an assignment an error shall occur if the referenced alternative is not the currently chosen alternative or if the referenced union field or value is omitted or uninitialized.

EXAMPLE 2:

```
type union MyUnion2
{
    integer choice1,
    charstring choice2
}
type record MyRecordEmbedsUnion
{
    MyUnion2 field1 optional
}
...
var MyUnion2 v_un2 := { choice1 := 1 }
var charstring v_char := v_un2.choice2; // causes an error as v_un.choice2 is not chosen
var MyRecordEmbedsUnion v_rec := { field1 := omit }
var integer v_int := v_rec.field1.choice1; // causes an error as v_rec.field1 is omitted
```

When referencing an alternative of a union type on the left hand side of an assignment, the referenced alternative shall become the chosen one. This rule shall apply recursively if the reference contains alternatives of nested unions, choosing all the referenced alternatives.

When referencing an alternative of an uninitialized union value or field or omitted field (including omitting a field at a higher level of the embedding hierarchy) on the left hand side of an assignment, the reference shall recursively be expanded up to and including the depth of the referenced alternative as follows:

a) When expanding a value or value field of **union** type, the alternative referenced in the dot notation becomes the chosen one.

b) Expansion of **record**, **record of**, **set**, **set of**, and **array** values and intermediate fields shall follow the rules of item a) in clauses 6.2.1.1 and 6.2.3, and clause 6.2.2.1 correspondingly.

c) At the end of the expansion, the value at the right hand side of the assignment shall be assigned to the referenced alternative.

EXAMPLE 3:

```
type union MyUnion3
{
    integer choice1,
    union
    {
        bitstring subchoice1,
        charstring subchoice2
    } choice2
}
...
var MyUnion3 v_un3 := { choice1 := 1 };
var MyRecordEmbedsUnion v_rec2 := { field1 := omit };
v_un3.choice2.subchoice2 := "Hello!";
// after the assignment v_un3 equals to { choice2 := { subchoice2 := "Hello!" } }
v_rec2.field1.choice1 := 10; // after the assignment v_rec2 equals to
                             // { field1 := { choice1 := 10 } }
```

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 v_myVarOne, v_myVarTwo;
var integer v_myVarThree;

v_myVarOne.integer := 34;
v_myVarTwo := {integer := v_myVarOne.integer + 1};

v_myVarThree := v_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 v_a1:= { 7, 8, 9 };
var MyRecordOfType1 v_r1:= v_a1;    // MyArrayType1 and MyRecordOfType1 are compatible

var integer v_myArray1[3]:= v_r1;   // Instantiates an integer array of 3 elements
                                    // with the index 0 to 2
                                    // being compatible to MyArrayType1 and MyRecordOfType1

var integer v_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 shall 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. An array or record of integer restricted to a single size can be used in the index notation as a short-hand for the repeated index notation. Accessing elements outside the array's range will cause a compile-time or test case error.

EXAMPLE 2:

```
v_myArray1[1] := 5;
v_myArray2[1][2] := 12;

v_myArray1[4] := 12;    // ERROR: index shall be between 0 and 2
v_myArray2[3][2] := 15; // ERROR: first index shall 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 syntax define the lower and upper index values. The upper value shall not be lesser than the corresponding lower value. 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 v_myArray3[1 .. 5]; // Instantiates an integer array of 5 elements
                                // with the index 1 to 5
v_myArray3[1] := 10; // Lowest index
v_myArray3[5] := 50; // Highest index

var integer v_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 index notation or more than one or all at once by a value list notation or index assignment notation. For using the value list or assignment notation for arrays, the rules described in clause 6.2.3 are valid for arrays as well.

Index 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 or if the index notation is applied to an uninitialized or omitted array value on the right hand side of an assignment, error shall be caused. Sending an array value with undefined elements shall cause an error. All elements in an array value that are not set explicitly are undefined. When referencing an element of an uninitialized array value or field or omitted field on the left hand side of an assignment, the rules for record of values specified in clause 6.2.3 apply.

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:

```
v_myArray1[0]:= 10;
v_myArray1[1]:= 20;
v_myArray1[3]:= 30;

// or using an value list
v_myArray1:= {10, 20, -, 30};

v_myArray4:= {{1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10}};
// the array value is completely defined

var integer v_myArray5[2][3][4] :=
{
    {
        {1, 2, 3, 4}, // assigns a value to v_myArray5 slice [0][0]
        {5, 6, 7, 8}, // assigns a value to v_myArray5 slice [0][1]
        {9, 10, 11, 12} // assigns a value to v_myArray5 slice [0][2]
    }, // end assignments to v_myArray5 slice [0]
```

```
    {
        {13, 14, 15, 16}, {17, 18, 19, 20}, {21, 22, 23, 24}
    } // assigns a value to v_myArray5 slice [1]
};

v_myArray4[2] := {20, 20};
    // yields {{1, 2}, {3, 4}, {20, 20}, {7, 8}, {9, 10}};
v_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}}};

v_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 v_myArrayInvalid[2][2];
v_myArrayInvalid := { 1, 2, 3, 4 }
    // causes an error as the dimension of the value notation
    // does not correspond to the dimensions of the definition
v_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 shall 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

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

a) At most one address type shall be bound to a port type.

b) At most one map parameter list shall be defined for a port type.

c) At most one unmap parameter list shall be defined for a port type.

d) Formal parameters of **map param** and **unmap param** declarations shall be value parameters and not be of **port**, **component**, **timer** or **default** type or of structured types having fields of **port**, **component**, **timer** or **default** type.

e) *MessageType* shall be a data type as defined in clause 3.1.

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 p_p1, out MsgType6 p_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 p_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.

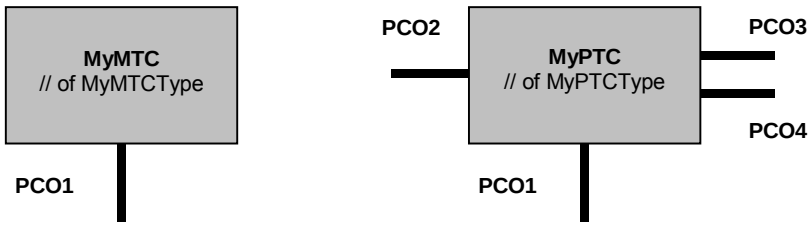


Figure 3: Typical components

It is also possible to declare constants, variables, templates 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, templates 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, templates or timers do not have to obey these restrictions.

Syntactical Structure

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

Semantic Description

Component type definitions specify the creation, declaration and initialization of ports and component constants, variables, templates 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, template 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 vc_myLocalInteger;
    timer tc_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
     { "," ComponentTypeIdentifier} "{"
     { ( PortInstance
     | VarInstance
     | TimerInstance
     | ConstDef
     | TemplateDef ) }
"}"
```

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, template, 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 vc_myLocalInteger;
    timer tc_myLocalTimer;
    port MyMessagePortType pCO1
}

type component MyExtendedMTCType extends MyMTCType
{
    var float vc_myLocalFloat;
    timer tc_myOtherLocalTimer;
    port MyMessagePortType pCO2;
}

// effectively, the above definition is equivalent to this one:
type component MyExtendedMTCType
{
    /* the definitions from MyMTCType */
    var integer vc_myLocalInteger;
    timer tc_myLocalTimer;
    port MyMessagePortType pCO1

    /* the additional definitions */
    var float vc_myLocalFloat;
    timer tc_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 vc_myLocalInteger;    // ERROR - already defined in MyMTCType (see above)
    var float tc_myLocalTimer;        // ERROR - timer with that name exists in MyMTCType
    port MyOtherMessagePortType pCO1; // ERROR - port with that name exists in MyMTCType
}

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

EXAMPLE 4: Component type extension from several parent types

```
type component MyCompB { timer tc_t };
type component MyCompC { var integer tc_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 vc_myVar1 := 10;
}

type component MyCompF extends MyCompB {
    var float vc_myVar2 := 1.0;
}

type component MyCompG extends MyCompB, MyCompE, MyCompF {
    // No name clash.
    // All three parent types of MyCompG have a timer tc_t, either directly or via extension of
    // MyCompB; as all these stem (directly or via extension) from timer tc_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 shall 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* shall 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 v_myCompInst := MyCompType.create;
```

EXAMPLE 2: Usage of component references in configuration operations

```
// referring to the component created above
connect(self:myPCO1, v_myCompInst:pCO1);
map(myCompInst:pCO2, system:extPCO1);
myCompInst.start(f_myBehavior(self));   // self is passed as a parameter to f_myBehavior
```

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

```
MyPCO1.receive from v_myCompInst;
 :
MyPCO2.receive(integer:?) -> sender v_myCompInst;
 :
MyPCO1.receive(mw_myTemplate) from v_myCompInst;
 :
MyPCO2.send(integer:5) to v_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
// MyCompType1, MyCompType2 and MyCompType3 and where the sender has to be retrieved.
// In this case the following scheme may be used:
 :
var M1 v_myMessage, v_myResult;
var MyCompType1 v_myInst1 := null;
var MyCompType2 v_myInst2 := null;
var MyCompType3 v_myInst3 := null;
 :
alt {
    [] pCO1.receive(M1:?) from MyCompType1:? -> value v_myMessage sender v_myInst1 {}
    [] pCO1.receive(M1:?) from MyCompType1:? -> value v_myMessage sender v_myInst2 {}
    [] pCO1.receive(M1:?) from MyCompType1:? -> value v_myMessage sender v_myInst3 {}
}
 :
v_myResult := f_myMessageHandling(v_myMessage); // some result is retrieved from a function
 :
if (v_myInst1 != null) {pCO1.send(v_myResult) to v_myInst1};
if (v_myInst2 != null) {pCO1.send(v_myResult) to v_myInst2};
```

```
if (v_myInst3 != null) {pCO1.send(v_myResult) to v_myInst3};
 :
```

EXAMPLE 5: Usage of self

```
var MyComponentType v_myAddress;
v_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 TC_MyTestCase() runs on MyMtcType system MyTestSystemInterface
{
     :
    var integer v_i;
    var MyPTCType1  v_myPtc[11];
     :
    for (v_i:= 0; v_i<=10; v_i:= v_i+1)
    {
        v_myPtc[v_i] := MyPTCType1.create;
        connect(self:ptcCoordination, v_myPtc[v_i]:mtcCoordination);
        v_myPtc[v_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 may 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.

c) The **`address`** data type shall not be used in the **`to`**, **`from`** and **`sender`** parts of receive and send operations of connected ports, i.e, ports used for the communication among test components.

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 v_mySUTentity := null;
 :
// receiving an address value and assigning it to variable MySUTentity
pCO.receive(address:?) -> value v_mySUTentity;
 :
// usage of the received address for sending template m_myResult
pCO.send(m_myResult) to v_mySUTentity;
 :
// usage of the received address for receiving a confirmation template
pCO.receive(mw_myConfirmation) from v_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 f_myFunction () runs on MyComponentType {
    var MyPortType.address v_sUT_Address := {5, true};  // address value for addressing
                                                        // via ports of MyPortType
    :
    pCO.send(integer: 5) to v_sUT_Address;              // use of address value in to
    :
    pCO.receive(integer: ?) from v_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 MyPortType1.address v_mySUTentity1 := null;  // type of v_mySUTentity1 is AddressType1
var MyPortType2.address v_mySUTentity2 := null;  // type of v_mySUTentity2 is AddressType2
var MyPortType3.address v_mySUTentity3 := null;  // type of v_mySUTentity3 is AddressType3
 :
// receiving address values and assigning them to variables
pCO1.receive(MsgType1:?) from address:? -> sender v_mySUTentity1;
                                                  // Address type of module scope,
                                                  // no prefix needed
pCO2.receive(MsgType2:?) from MyPortType2.address:? -> sender v_mySUTentity2;
                                                  // Resolution of address type
                                                  // by means of a prefix
pCO3.receive(MsgType3:?) from MyPortType3.address:? -> sender v_mySUTentity3;
 :
// usage of the received address values for addressing purposes
pCO1.send(v_myResult) to v_mySUTentity1;
 :
pCO2.receive(mw_myConfirmation) from v_mySUTentity2;
 :
pCO3.send(m_myRequest) to v_mySUTentity3;
 :
```

6.2.13 Subtyping of structured types

6.2.13.0 General

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 number of elements for 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. In case of the range syntax the upper bound shall not be lesser than the lower bound value.

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 matched by at least one of the constraints in the list. In case of list subtyping of **`record`**, **`set`**, **`record of`**, **`set of`**, **`union`** and **`anytype`** types, the list may contain both subtypes and possibly partial templates of the parent types. Subtype references shall be resolved in a recursive way: the collection of templates denoted by the subtype(s) referenced in the list become members of the new subtype definition with an expanded list containing only possibly partial templates. When constraining **`record of`**, **`set of`**, **`union`** and **`anytype`** types, all templates of the expanded list (i.e. after resolving the subtype references) shall be valid (i.e. complete) templates of the first parent type. When constraining **`record`** and **`set`** types, templates of the expanded list defined using the value list notation shall be valid (i.e. complete) templates, while templates of the expanded list defined using the field assignment notation may be partial (i.e. incomplete). In the latter case, the fields that are not explicitly present shall be considered as containing *AnyValue* for mandatory fields and *AnyValueOrNone* for optional fields.

NOTE: Users should assign new values to single fields of values/templates based on types using list subtyping cautiously: it may happen that the new field value would be valid with other combination(s) of the rest of the fields but causes an erroneous record/set value, when combining with the actual values of the other fields. See example 1 below.

In case of **`enumerated`** types, the template 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 of 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" }
) // invalid type as { f1 := 1, f2 := "user", f3 := "password" } is not a legal value of
  // MyRecordSub1 (notice field f1)

type MyRecord MyRecordSub4 (
  { f2 := "user", f3 := "password" },
  { f2 := "User", f3 := "Password" }
) // any valid value of MyRecord, where the combination of f2 and f3 is
  // f2 := "user" AND f3 := "password" or f2 := "User" AND f3 := "Password"
  // i.e. field f1 is considered as if it was present and contained AnyValueOrNone

type MyRecord MyRecordSub5 (
  { f2 := "user", f3 := pattern "password|Password" },
  { f1 := (1 .. 10), f2 := "User" }
) // a valid subtype of MyRecord containing all values which match one of the given
  // templates
  // { f1 := *, f2 := "user", f3 := pattern "password|Password" } or
  // { f1 := (1 .. 10), f2 := "User", f3:= ? }

type record R { integer k, integer i, integer j }
type R R2 ({ k:= 1, i := 2}, { k:= 2, i := 3})

function f_inc(inout integer p_p) {
  p_p := p_p + 1;
}

function f() {
  var R2 v_x := { 1, 2, 5 }
  v_x.k := 2; // error, as the value {2,2,5} is not allowed
  inc(v_x.i); // error, as the value {1,3,5} is not allowed
              // (previous erroneous assignment is ignored here)
  inc(v_x.j); // allowed
}
```

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 { e_first, e_second, e_third, e_fourth, e_fifth };

type MyEnum EnumSub1 ( e_first, e_second, e_third );
  // a valid subtype of MyEnum

type EnumSub1 EnumSub2 ( e_first, e_second );
  // a valid subtype of EnumSub1

type EnumSub1 EnumSub3 ( e_first, e_second, e_fourth );
  // causes an error as e_fourth is not a value of EnumSub1

type MyEnum EnumSub4 ( EnumSub1, e_fourth);
  // causes an error as type references are not allowed in the template 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

type record R { integer k, integer i, integer j }
type R R2 ({ k:= 1, i := 2}, { k:= 2, i := 3})

function f_g() {
  var R2 v_x := { 1, 2 }
  v_x.k := 2; // error
}
```

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 String23Array2D56 length(2..3);
// is an unordered two-dimensional array of the size 5*6 strings, each composed
// of 2 or 3  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 String23Array2D56 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 String23Array2D56 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

6.3.0 General

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

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

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 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". Compatibility between charstring and universal charstring is defined below.

EXAMPLE 1: Compatibility of integers

```
// Given
type integer MyInteger(1 .. 10);
:
var integer v_x;
var MyInteger v_y;

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

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

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

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

EXAMPLE 2: Compatibility of floats

```
// Given
type float PositiveFloats(0.0 .. infinity);
:
var PositiveFloats v_x;
var float v_y;

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

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

v_y := not_a_number; // is a valid assignment
v_x := v_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 v_myCharacter;
var charstring v_myCharString;
var MySingleChar v_mySingleCharString := "B";

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

//Given
v_myCharString := "abcd";

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

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

//Then
v_myCharString := v_myCharacterArray[1];
//is valid and assigns the value "B" to v_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".

EXAMPLE 4: Compatibility of character and universal character strings

```
//Given
type charstring MyChar length (1);
...
var MyChar v_myCharacter;
var charstring v_myCharString;
var universal charstring v_myUnivCharString;


// Given
v_myCharString := "abcd";
```

```
// Then
v_myUnivCharString := v_myCharString
//is valid as charstring and universal charstring are compatible
v_myCharacter := v_myUnivCharString [1];
// is valid as the r.h.s. notation addresses a single element of the string,
// containing a character compatible with charstring

// Given
v_myUnivCharString := "bet" & char ( 0, 0, 1, 113);

// Then
v_myCharString := v_myUnivCharString;
// is invalid as v_myUnivCharString contains a character not in ISO 646.

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

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

6.3.2 Compatibility of structured types

6.3.2.0 General

This clause defines compatibility rules for structured types. In subsequent clauses, "value "b"" is called the value to be assigned, e.g. when passed as parameter, to an object of type "A".

6.3.2.1 Compatibility of enumerated types

Enumerated types are only compatible with other **enumerated** types. An enumerated value "b" of an enumerated type "B" is compatible with enumerated type "A" if the identifier of the value "b" is also defined in "A" and the integer(s) associated with value "b" are also associated with the same identifier in "A".

6.3.2.2 Compatibility of record and record of types

record types 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 v_myVarA := { -, 1, true};
var BType v_myVarB := { omit, 2, true};
var CType v_myVarC := { 3, omit, true};
var DType v_myVarD := { 4, 4, true};
var EType v_myVarE := { 5, 5, true, omit};

// Then

v_myVarA := v_myVarB;   // is a valid assignment,
                        // new value of MyVarA is ( a :=omit, b:= 2, c:= true)
v_myVarC := v_myVarB;   // is a valid assignment
                        // new value of MyVarC is ( d :=omit, e:= 2, f:= true)
v_myVarA := v_myVarD;   // is NOT a valid assignment because the optionality of fields does not
                        // match
v_myVarA := v_myVarE;   // is NOT a valid assignment because the number of fields does not match

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

record of types and arrays are compatible if their element types are compatible and value "b" does not violate any length subtyping of the **record of** type "A" or dimensions of the array type. 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.

EXAMPLE 2:

```
// Given

type record of integer IType;

type record of float HType;

var HType v_myVarH := { 1, omit, 2};
var IType v_myVarI;
var integer v_myArrayVar[2];

// Then

v_myVarI := { 3, 4 };
v_myArrayVar := v_myVarI;
// is a valid assignment as element types are compatible and the assigned value
// doesn't violate length restriction set by array dimension

v_myVarI2 := v_myArrayVar;
// is a valid assignment as element types are compatible and the target variable type has
// no length restriction

v_myVarI[2] := 5; // the value of v_myVarI is { 3, 4, 5 } now
v_myArrayVar := v_myVarI;
// is NOT a valid assignment as v_myVarI contains more elements than the array dimension
// allows

v_myVarH := v_myVarI;
// is NOT a valid assignment as element types are not compatible
```

6.3.2.3 Compatibility of set and set of types

set types are only compatible with other **set** types and **set of** types are only compatible with other **set of** types. For **set** types the same compatibility rules shall apply as to **record** types and for **set of** types the same compatibility rules shall apply as to **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 v_myVarF := { a:=1, c:=true };
var GType v_myVarG := { f:=true, d:=7};

// Then

v_myVarF := v_myVarG;   // is a valid assignment as types FType and GType are compatible

v_myVarF := v_myVarA;   // is NOT a valid assignment as v_myVarA is a record type
```

6.3.2.4 Compatibility of union types

The compatibility rules for **union** types are are the following:

- A union value "b" of union type "B" is compatible with union type "A" if the alternative selected in "b" has a corresponding alternative with identical name in "A" and the value of the selected alternative in "b" is compatible to the type of the corresponding alternative in "A".
- Otherwise, the following rules apply. A **union** value "b" of **union** type "B" with a default alternative of type "C" is compatible with an arbitrary type "A" if the alternative selected in "b" is the default alternative and the value of the default alternative is compatible to "A". A value "a" of an arbitrary type "A" is compatible with a **union** type "B" with a default alternative of type "C" if value "a" is compatible to "C".

When considering the compatibility of two **union** types, initially the first rule (which is not dependent on the existence of a default alternative) shall be applied. The second rule shall only be used to check compatibility, when - using the first rule - no compatibility has been determined. This order shall avoid ambiguity in case that a default alternative would otherwise also be compatible with the union itself.

NOTE 1: It is possible to have nested unions with default alternatives. The rules above make type compatibility along the default alternatives alternatives transitive, i.e. the outermost **union** type is compatible with the type of the innermost default **union** alternative if all containing alternatives are also default alternatives.

NOTE 2: When a **union** with a default alternative is used in an expression it will be resolved to its long notation, before the expression is evaluated.

EXAMPLE 1:

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

var U1 v_u1 := {i := 1};
var U2 v_u2 := v_u1;        // correct
v_u1:= v_u2;                // correct as the alternative i is selected in v_u2 and is
                            // compatible to i in U1
v_u2:= {b := true};
v_u1:= v_u2;                // incorrect as v_u1 has no alternative b
var anytype v_x := v_u1;    // incorrect as the anytype is not a union type.
```

EXAMPLE 2: Using union values of unions with default alternatives

```
type union U3 { @default integer i, boolean b }
type union U4 { integer i, @default boolean b }
var U3 v_u3 := 3            // correct as i in U3 is declared with @default
```

```
v_u3 := v_u2;           // correct because all alternatives in U2 exist in U3
                        // and are compatible
v_u2 := 3;              // incorrect as 3 is not of a union type and there is
                        // no field in U2  declared with @default
v_u2 := v_u3;           // also correct
v_u2 := v_u1.i;         // incorrect
v_u3 := v_u1.i;         // correct

var integer v_int := v_u3 * 2   // v_int is 6 as v_u3 is treated as v_u3.i

var U3 v_u32 := {b := true};
var U4 v_u4 := true;
v_int := v_u4 *2;       // incorrect as v_u4 is treated as a boolean, and can not be multiplied
v_int := v_u32 *2;      // test case error as "v_u32" would be treated as v_u32.i,
                        // which is not the selected alternative
log(v_u4);              // results in "{ b:=true}" logged; for backward compatibility when
                        // a union value is used in a log statement directly, no conversion
                        // is performed.
```

6.3.2.5 Compatibility of anytype types

anytype types are only compatible with other **anytype** types. An anytype value "b" of anytype type "B" is compatible with anytype type "A" if the alternative selected in "b" is also contained in "A".

NOTE: Only anytype types that are constrained to a fixed set of types via list subtyping can be a potential cause for anytype incompatibility, i.e. if the set of types contained in type "A" does not contain the type selected in "b".

EXAMPLE:

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

module B {
    type integer I (0..2);
    type anytype Atype ({I:=?},{integer:=?},{float:=?});
}

module C {
    import from A all;
    import from B all;
    type union U {
        integer I (0..2)
    }
    control {
        var A.Atype v_aa;
        var A.Atype v_aaI := { A.I := 1 } // type I is is imported from A and B
        var A.Atype v_aaF := { F := 1.0 } // type F is only imported from A
        var B.Atype v_ba := { integer := 1 }
        var B.Atype v_baI := { B.I := 1 } // type I is imported from A and B
        var U v_u := { I := 1 } // I is a field name in U

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

    v_ba := { float := 1.0 };   // correct, assigning a literal value
    v_ba := v_aaI;      // incorrect, type A.I is not present in the anytype B.Atype
    v_ba := v_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 {
    AType   a,
    integer b optional,
    integer c
}

var JType v_myVarJ;

// If considering the declarations above, then


v_myVarJ.a := v_myVarA;
// is a valid assignment as the type of field a of JType and AType are compatible

v_myVarB := v_myVarJ.a;
// is a valid assignment as BType and the type of field a of JType are compatible
```

6.3.3 Compatibility of component types

Type compatibility of component types has to be considered in 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 altstep 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".

3) Mtc compatibility: a function or altstep referring to component type "A" in its mtc clause may be called or started in any context that has a mtc clause of type "B" or a testcase with a runs on clause of type "B" if all the port definitions of "A" have identical definitions in "B". If the type of the mtc is unknown in the calling function, this can lead to runtime errors if the component type "A" is not mtc-compatible with the type of the running mtc.

4) System compatibility: a function or altstep referring to component type "A" in its system clause may be called or started in any context that has a system clause of type "B" or a test case with a runs on clause of type "B" and no system clause if all the port definitions of "A" have identical definitions in "B". If the type of the system is unknown in the calling function, this can lead to runtime errors if the component type "A" is not system-compatible with the type of the system the current test case was started on.

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 and connection operations

The communication operations (see clause 22) **send**, **receive**, **trigger**, **call**, **getcall**, **reply**, **getreply** and **raise** and connection operations **connect, map, disconnect** and **unmap** (see clause 21.1) are exceptions to the weaker rule of type compatibility and require strong typing. The types of values or templates directly used as parameters of the operations **send**, **receive** and **trigger** shall also be explicitly defined in the associated port type definition. The signature type of the parameter list given to the operations **call**, **getcall**, **reply**, **getreply** and the signature type given to the operations **catch** and **raise** shall also be explicitly defined in the associated port type definition. The types of values or templates directly used as exceptions to the operations catch and raise shall be explicitly defined in the **exceptions** part of the definition of the signature given to the 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 m_myRecAlias:= {...}         // a template of the alias type

var MyComponent v_myComp1 := MyComponent.create, v_myComp2 := MyComponent.create;
connect (v_myComp1:p, v_myComp2:p)               // two connected PTCs via ports that can
                                                 // transport the user-defined and the alias type

// in v_myComp1:
p.send (m_myRecAlias);                           // sending of template of alias type

// in v_myComp2:
p.receive (MyRec:?);
// shall not match as the transmitted template is of the alias type and
// not of the user-defined type

// in v_myComp2:
var MyRec v_x;
p.receive (MyRecAlias:?) -> value v_x;
// shall not cause an error since storing the value requires no 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

7.0 General

TTCN-3 allows the specification of expressions. TTCN-3 expressions may be template references, value references or literals (i.e. no operation is involved), and may be composed of the operators defined in clause 7.1.

NOTE: Templates can be used at the RHS of assignment, parameter passing and (predefined) functions where template passing is explicitly allowed.

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.

Assignment or list notations are used for expressions of record, set, record of, set of, array, union and anytype types.

Restrictions

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

a) At the point, when an expression is evaluated, the evaluated values of the operands used in expressions shall be completely initialized except where explicitly stated otherwise in the specific clause of the operator.

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

c) With the exception of the equality and non-equality operators, the special value **`null`** shall not be used as an operand of expressions (see clause 7.1.3).

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

```
(c_x + c_y – f_increment(c_z))*3    // single expression
{ a:= 1, b:= true }                 // compound expression, assignment notation
{ 1, true }                         // compound expression, list notation
```

7.1 Operators

7.1.0 General

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

a) arithmetic operators;

b) 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) When an expression is evaluated, the evaluated values used as the operands of operators shall be completely initialized, except for those operands for which it is explicitly allowed to be partially initialized (see clause 11.1).

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.

The usage of the special float values **infinity**, **-infinity** and **not_a_number** in arithmetic operators shall follow the rules defined in IEEE 754™ [6].

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 1: 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.

NOTE 2: It is also possible to concatenate two or more value list notation expressions if the result is to be used as a **`record of`** or **`array`** of the same root type as the concatenated expressions.

Restrictions

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

a) When the list concatenation operator is used for record of-s, set of-s and arrays, its operands shall be at least partially initialized.

EXAMPLE:

```
'1111'B & '0000'B & '1111'B gives '111100001111'B
{1,2} & {3,4} & {5,6} gives the following record of integer {1,2,3,4,5,6}
```

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.

The **`address`** type is allowed for the equality (==) and non-equality (!=) operators, independent of its actual type, but when its actual type differs from the types specified above, it can be compared to the literal special value **`null`** only.

Operands of equality (==) and non-equality (!=) shall be completely initialized values or field references of type compatible root types and the values or field references 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.

- Two field references are equal if the referenced fields are both **`optional`** fields and both fields are set to **`omit`** or if both referenced fields (regardless if they are optional or not) are initialized with values and these values are equal. A field reference is equal to a value if the referenced field is initialized with a value and both values are equal.
- Two integer values are equal if and only if they contain the same value. Otherwise, normal mathematical ordering is applied.
- Two enumerated values are equal if and only if they are associated with the same integer value. Otherwise, they are ordered using the mathematical order on the associated integer values.

- 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 float 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, or set values are equal respectively if and only if they are mutually compatible with the type of the other operand (see clause 6.3), the actual values of all present fields are equal to their corresponding fields and all fields corresponding to omitted fields are also omitted in the peer value.

- Two record of values, set of values or array values, respectively, are equal if and only if they are mutually compatible with the type of the other operand (see clause 6.3), they both have the same length, and and each element of one value is equal to the corresponding element of the other value. Record of values and array values may also be compared, in which case the corresponding record of type of the array is being considered.

- 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 user-defined types defined with the same name in different modules do not denote the same type 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).

- It is also possible to use compound expressions (field assignment or value list notation) directly as operands of comparison operations of structured types. If there is a compound expression on both sides of the comparison operator, they shall both be value list notation expressions where the elements shall be of the same root type and they shall be compared like record of values with elements of that root type. If only one operand of the comparison operation is a compound expression it shall be compatible with the root type of the other operand and they shall be compared like values of that root type.

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  c_s1    :=  { a1 := 0, a2 := omit, a3 := 2 };
    // Notice that the order of defining values of the fields does not matter
const   S2  c_s2a   :=  { b1 := 0, b3 := 2, b2 := omit };
const   S2  c_s2b   :=  { b2 := 0, b3 := 2, b1 := omit };
const   S3  c_s3    :=  { c1 := 0, c2 :=2 };
var     SI  v_si:=  { 0, -, 2 };
const   SI  c_si    :=  { 0, 2 };
const   U1  c_u1    :=  { d1:= 0 };
const   U2  c_u2    :=  { e1:= 0 };
const   U3  c_u3;   :=  { d1:= 0 };

// Then
c_s1 == c_s2a;
    // returns true
c_s1 == c_s2b;
    // returns false, because neither a1 nor a2 are equal to their counterparts
    // (the corresponding element is not omitted)
c_s1 == c_s3;
    // returns false, because the effective value structures of s1 and s3 are not compatible
c_s1 == v_si;
    // causes test case error as v_si is not completely initialized
    // (2nd element is left uninitialized)
c_s1 == c_si;
    // returns false, as the counterpart of the omitted a2 is 2,
    // but the counterpart of a3 is undefined
c_s3 == c_si;
    // returns true
c_u1 == c_u2;
    // causes error as U1 and U2 have no common subset of alternatives
c_u1 == c_u3;
    // returns true, as alternatives with the same names are chosen and
    // the actual values in the selected alternatives are equal
{ 0, omit, 2 } == c_s1;
    // returns true
c_s2a == { b1 := 0, b2:= omit, b3 := 2 };
    // returns true
{ c_s1, c_s2b } == { c_s2a, c_s1 };
    // returns false because c_s2b != c_s1
{ c_s1, c_s2b, c_s2a } == { c_s1 };
   // returns false because of different length
c_s1.a1 == c_s2a.b1;
   // returns true, both fields are initialized with values and the values are equal
c_s1.a2 == c_s2a.b2;
   // returns true, both fields are omit
c_s1.a1 == c_s2a.b2;
   // returns false, value vs. omit
c_s1.a1 == omit;
   // error, omit is neither a value nor a field reference
c_s1.a2 == 3;
   // false, omit vs. value
```

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 1: 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.

Restrictions

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

a) When the rotate operator is used for **record of**-s, **set of**-s and arrays, its left hand operand shall be at least partially initialized.

NOTE 2: Please note that for the right hand operand restriction a) in clause 7 further on applies.

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.

7.3 Decoded field reference

Decoded field reference is a specific notation that can be applied to expressions of **bitstring**, **hexstring**, **octetstring**, **charstring** or **universal charstring** types. It is used for accessing content of implicitly decoded payload fields.

Syntactical Structure

```
ReferencedValue "=>" ( PredefinedType | TypeIdentifier |
     ( "(" Type [ "," Expression ] ")" ) )
```

The string value preceding the **=>** operator shall be decoded into a value of the type following the **=>** operator. Failure of this decoding shall cause a test case error. In case the string operand is of the **universal charstring** type and the extended syntax with parentheses is used, the type operand can be followed by an optional parameter defining the encoding format. The parameter shall be of the **charstring** type and it shall contain one of the strings allowed for the **decvalue_unichar** function (specified in clause C.5.4). Any other value shall cause an error. In case the string operand is not a **universal charstring**, the optional parameter shall not be present.

Restrictions

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

a) If the type operand is not enclosed into parentheses, only a built-in type or a type reference consisting of a single identifier can be used. Extended type references shall always use the extended syntax with parentheses.

EXAMPLE:

```
type record PDU {
    PduHeader header,
    bitstring outerPayload
}

type record OuterPayload {
    OuterPayloadHeader header,
    universal charstring innerPayload
}

type record InnerPayload {
    integer data1,
    charstring data2
}

...
var PDU v_pdu;
var InnerPayload v_inner;
... // v_pdu is filled with data;
v_inner := v_pdu.outerPayload=>OuterPayload.innerPayload=>(InnerPayload, "UTF-8");
// v_pdu.outerPayload field is first decoded into a value of the OuterPayload type
// the innerPayload field of the decoding result is subsequently decoded into a value
// of the InnerPayload type (using UTF-8 format for conversion into a bitstring)
```

8 Modules

8.0 General

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 version 4.3.1 of the present document (see annex H).
`"TTCN-3:2012"` - to be used with modules complying with version 4.4.1 of the present document (see annex H).
`"TTCN-3:2013"` - to be used with modules complying with version 4.5.1 of the present document (see annex H).
`"TTCN-3:2014"` - to be used with modules complying with version 4.6.1 of the present document (see annex H).
`"TTCN-3:2015"` - to be used with modules complying with version 4.7.1 of the present document (see annex H).
`"TTCN-3:2016"` - 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 ETSI ES 202 781 [i.11], ETSI ES 202 782 [i.14], ETSI ES 202 784 [i.12], and ETSI ES 202 785 [i.13]. The language identifier and the package identifier are to be written as a comma-separated list.

Restrictions

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

a) At most one language string per module shall be given to define the core language version in which the module is defined.

b) Per extension package, at most one extension package string of that extension package shall be used by a module.

Examples

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

8.2 Module definitions part

8.2.0 General

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 runtime. 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 runtime. For module parameterization, TTCN-3 only supports value parameterization which has to be resolved static at start of runtime.

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 runtime value for a module parameter, the default value shall be used during test execution, otherwise the actual value provided by the test system. Actual runtime values shall be literals only.

If functions are used for the initialization of module parameters, it is strongly advised to adhere to the rules defined in clause 16.1.4. Not following these rules may cause non-deterministic test executions.

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 and shall not be of a structured type that contains a sub-element of **`port`** type at any level of nesting.

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.

h) All sub-elements of **`component`** or **`default`** type of a default value of a module parameter shall be initialized with the special value **`null`**.

Examples

```
module MyTestSuiteWithParameters
{
    // single type, single module parameter, which is per default public
    modulepar boolean PX_Par0 := true;

    // single type, multiple module parameters with an explicit public visibility
    public modulepar integer PX_Par1, PX_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 c_myConst:= 1;
     :
        type record MyMessageType { … };
        group myGroup1 {    // Sub-group with definitions
            type record AnotherMessageType { … };
            const boolean c_myBoolean := false
        }
    }

    // A group of altsteps
    group myStepLibrary {
        group myGroup1 {    // Sub-group with the same name as the sub-group with definitions
            altstep a_myStep11() { … }
            altstep a_myStep12() { … }
             :
            altstep a_myStep1n() { … }
        }
        group myGroup2 {
            altstep a_myStep21() { … }
            altstep a_myStep22() { … }
            :
            altstep a_myStep2n() { … }
        }
    }
    :
}

// An import statement that imports myGroup1 within myStepLibrary
import from MyModule {
    group myStepLibrary.myGroup1
}
```

8.2.3 Importing from modules

8.2.3.0 General

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	f_myFunction	(**inout** MyType1 p_myPar) **return** MyType2
		runs on MyCompType
			{
			  const MyType3 c_myConst := …;
			  : // further behaviour
			}

	Specification	Name	Specification

type	**record**	MyRecordType	{
			  MyType4 field1,
			  integer field2
			}

	Specification	Name	Specification

template	MyType5	m_myTemplate	:= {
			  field1 := 1,
			  field2 := c_myConst, // c_myConst is a module constant
			  field3 := PX_ModulePar // PX_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	f_myFunction	p_myPar	MyType1, MyType2, MyCompType
type	MyRecordType	field1, field2	MyType4, integer
template	m_myTemplate		MyType5, field1, field2, field3, c_myConst, PX_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 c_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 ("."). If the type of the component referenced in a connection operation is known (either when the component reference is a variable or value returned from a function or the type is defined the runs on, mtc or system clause of the calling function), the referenced port declaration shall be present in this component type.

There is one exception to this rule: when **in the context** of an enumerated type (see clause 6.2.4), an enumerated value is clashing with the name of a definition in the importing module, the enumerated 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. For the latter case, prefixing shall only be used for definitions with global visibility for the module.

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 an enumerated 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 enumerated values are known when an 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 (see clause 8.1) 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 f_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 c_myConst := …;

    template RecordType_T2 m_t2 (RecordType_T1 p_tempParT2):= { // parameterized template
        Field1_T2 := …,
        :
    }

} // end module ModuleONE


module ModuleTWO {

    import from ModuleONE {
        template m_t2
    }

    // Only the names m_T2 and p_tempParT2 will be visible in ModuleTWO. Please note, that
    // the identifier p_tempParT2 can only be used when modifying m_t2. All information
    // necessary for the usage of m_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 c_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 v_myComponentVar := ModPar2;
                                // Reference to a module parameter of ModuleONE
        :
    }

    function f_myFunction () return integer {
        return c_myConst        // Reference to a module constant of ModuleONE
    }

    testcase TC_MyTestCase (out RecordType_T2 p_myPar) runs on MyCompType {

        :
        MyPort.send(m_t2); // Sending a template defined in ModuleONE
        :

    }

} // end ModuleTHREE
```

```
module ModuleFOUR {

    import from ModuleTHREE {
        testcase TC_MyTestCase
    }

    // Only the name TC_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 TC_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 v_myVar1 := "Test String";  // Prefix shall be used
        var MyTypeA v_myVar2 := '10110011'B;                // This is the original MyTypeA
         :
        var MyTypeB v_myVar3 := "Test String";              // Prefix need not be used …
        var SomeModuleC.MyTypeB v_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 enumerated values and global definitions

```
module A {
  type enumerated MyEnumType {enumX, enumY}
  type enumerated MyEnumType2 {enumY, enumZ}
}

module B {
  import from A all;
  const MyEnumType enumY := enumX; // this is not allowed as enumerated values restrict
                                   // global names (see clause 6.2.4)

  const MyEnumType2 enumX := enumY;// this is likewise not allowed

  const MyEnumType enumZ := enumX; // allowed as MyEnumType does not contain enumZ
}

module C {
  import from A all;
  import from B all;

  const integer enumZ := 0;
  const integer enumY := 1;
  const MyEnumType2 enumX := enumY;


  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 enumerated values
  // take precedence over global definition names; note that for the same context reason there
  // is no name clash between the enumerated values defined in MyEnumType and in MyEnumType2

  modulepar MyEnumType PX_MyModulePar2 := B.enumZ
  // the default value of the module parameter will be the value enumX, as the prefix
  // identifies the constant definition enumZ 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 := C.enumX
  // causes an error as PX_MyModulePar3 and the constant enumX in module C has different types
}
```

EXAMPLE 5: Importing local definitions transitively

```
module A {
  type enumerated MyEnumType { 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 MyEnumType PX_MyModulePar := enumY;
  type component MyCompUser extends MyComp {}
}

module C {
  import from B all;
  testcase TC() runs on MyCompUser {
    if (PX_MyModulePar == enumY) {
      // the enumerated value 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 m_myTemplate1;             // imports one template, and
    const c_myConst1, c_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 { QualifiedIdentifier [ 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     { QualifiedIdentifier   [ "," ] } | 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 (see clause 8.1) 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 ETSI ES 202 781 [i.11], ETSI ES 202 782 [i.14], ETSI ES 202 784 [i.12] and ETSI 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 ETSI ES 201 873-7 [i.5], ETSI ES 201 873-8 [i.6] and ETSI 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 or 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.

When importing definitions from a non-TTCN-3 language, two principle approaches exist:

- With an implicit language mapping, non-TTCN-3 definitions are mapped internally in the TTCN-3 tool to the respective TTCN-3 definitions as defined by the language mapping; the importing module works with the internal representations of the imported definitions.
- With an explicit language mapping, non-TTCN-3 definitions are mapped directly to separate TTCN-3 definitions; the importing module imports the generated TTCN-3 and works with the mapped TTCN-3 definitions.

These lead to three options when using non-TTCN-3 language modules in a TTCN-3 specification:

- The import statement imports the non-TTCN-3 module; the tool uses the internal representation of the implicit mapping of the non-TTCN-3 module's definitions according to the language mapping specification of that language.
- The import statement imports the non-TTCN-3 module; the tool imports from a TTCN-3 module which is an explicit mapping of the non-TTCN-3 module's definitions according to the language mapping specification of that language.

- The import statement imports the explicit TTCN-3 representation of the non-TTCN-3 module; the tool imports the TTCN-3 module which is an explicit mapping of the non-TTCN-3 module according to the language mapping specification of that language.

Restrictions

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

a) The language specification should 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 ETSI ES 201 873-7 [i.5], ETSI ES 201 873-8 [i.6] or ETSI 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 (see clause 8.1) 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 mw_t1 := ?;
    template T2 mw_t2 := *;
    :
}
module B {
    public import from A { type T1 }
    type charstring T2;
    template T1 m_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.mw_t1  (defined)
// A.mw_t2  (defined)
//
// module B knows
// A.T1     (imported)
// B.T2     (defined)
// B.m_t1   (defined)
//
// module C knows
// A.T1     (imported from B importing it from A)
// B.T2     (imported)
// A.mw_t1  (imported)
// A.mw_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.mw_t1 and A.mw_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 (see clause 8.1) 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) A TTCN-3 module shall only import from earlier or same editions of TTCN-3 but not from later editions, e.g. the TTCN-3 language specification in an import statement has to be lower or equal to the TTCN-3 language specification of the importing module.

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 c_myConstant := 1;
    type record MyMessageType { … }
    template MyMessageType m_myMessage := { … }
    :
    function f_myFunction1() { … }
    function f_myFunction2() { … }
    :
    testcase TC_MyTestcase1() runs on MyMTCType { … }
    testcase TC_MyTestcase2() runs on MyMTCType { … }
    :
    // … and a control part so it is executable
    control
    {
        var boolean v_myVariable; // local control variable
        :
        execute( TC_MyTestCase1()); // sequential execution of test cases
        execute( TC_MyTestCase2());
        :
    }
}
```

9 Port types, component types and test configurations

9.0 General

TTCN-3 allows the (dynamic) specification of concurrent test configurations (or configuration for short).
A configuration consists of a set of inter-connected test components with well-defined communication ports and an explicit test system interface which defines the borders of the test system (see figure 4).

NOTE: Additional configuration and deployment support for TTCN-3 is defined in the optional package [i.11].

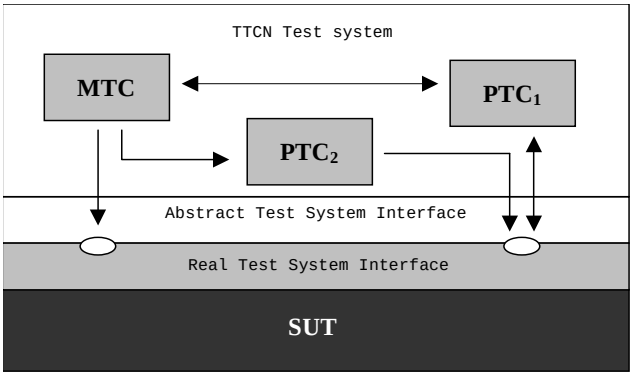


Figure 4: Conceptual view of a typical TTCN-3 test configuration

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).

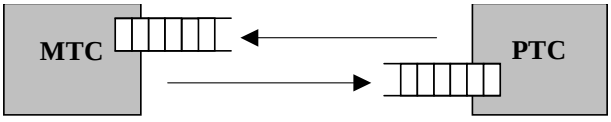


Figure 5: The TTCN-3 communication port model

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 (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 (figure 7 (a) and (e)).

- A port of a test system interface cannot have connection with more than one port owned by a component A. This means, connections as shown in figure 7 (b) are not allowed**.**
- 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 figure 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 runtime and shall lead to a test case error when failing.

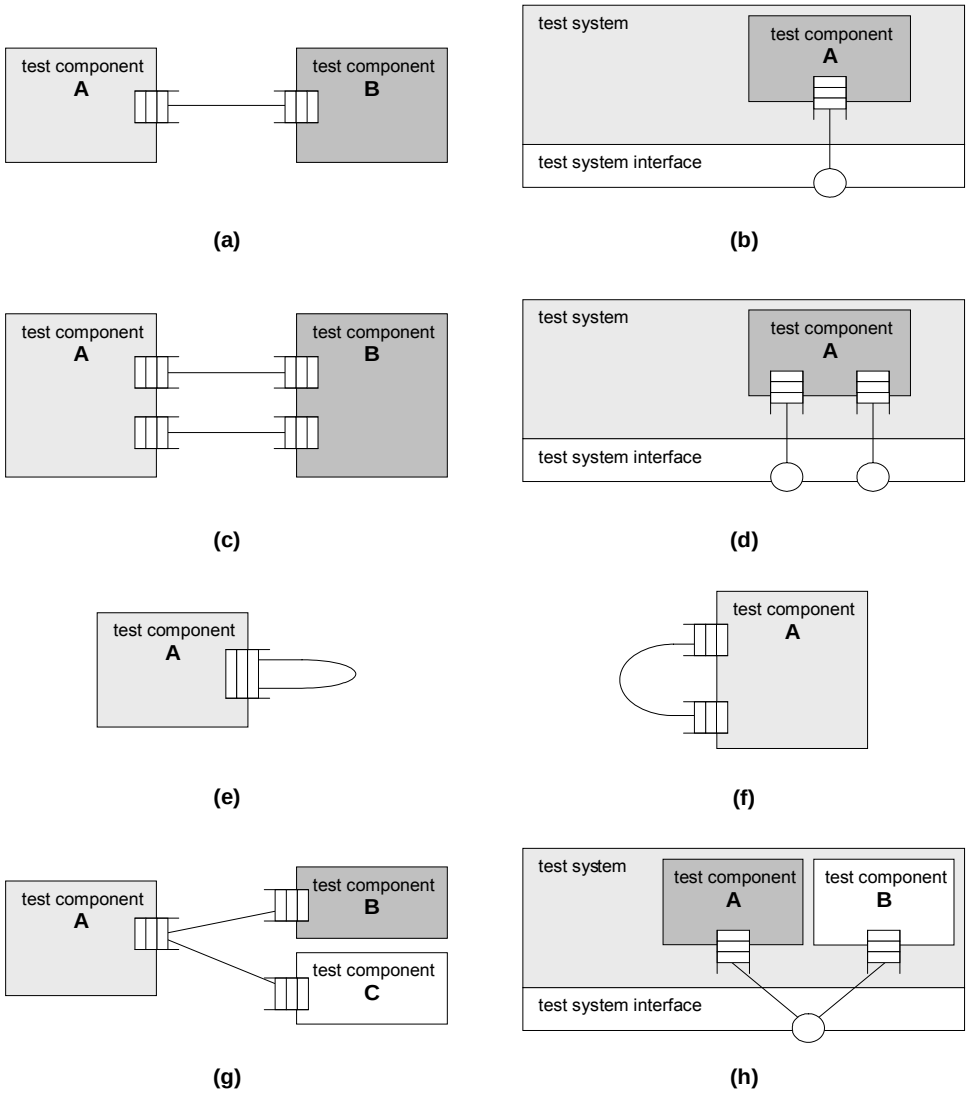


Figure 6: Allowed connections

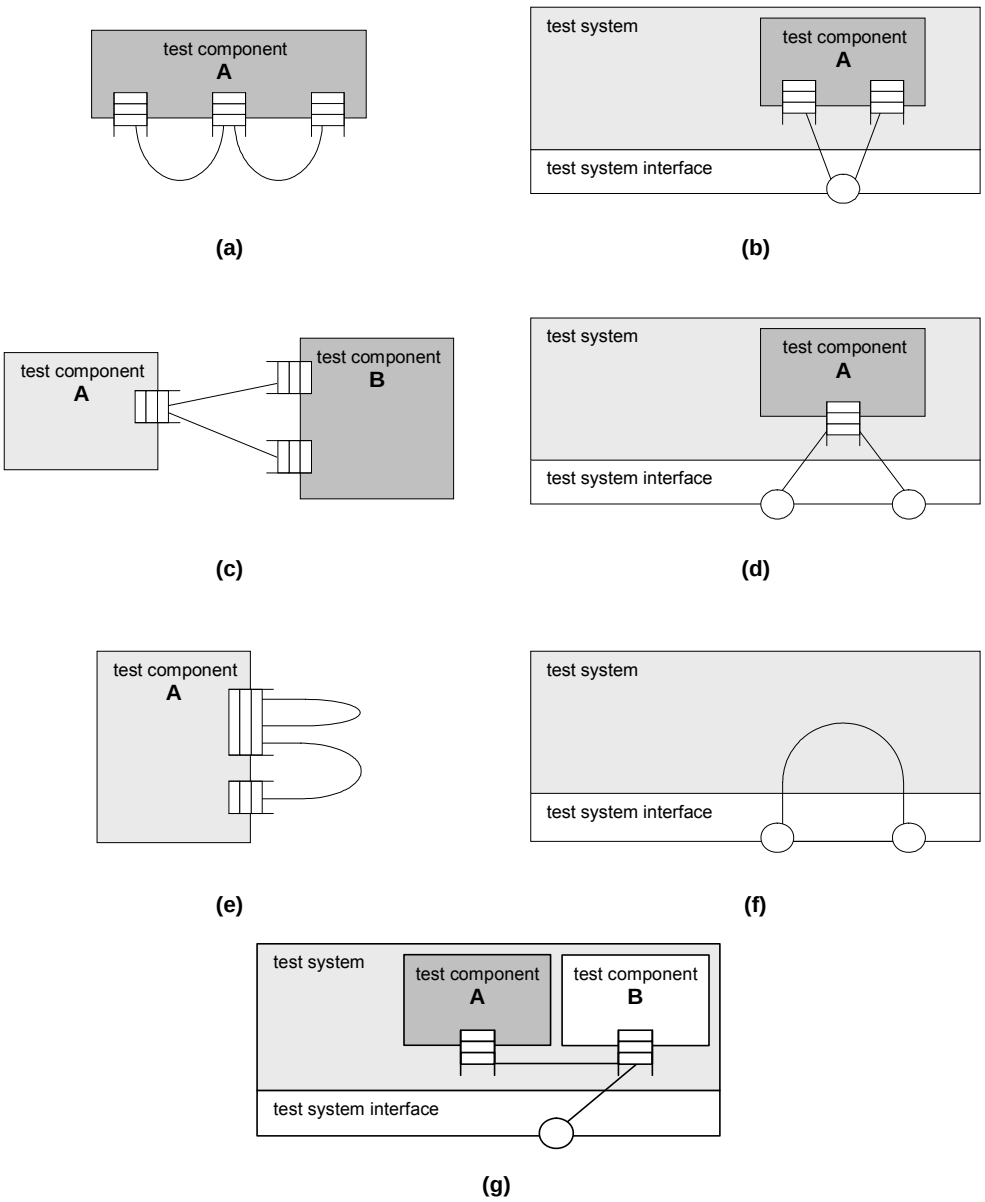


Figure 7: NOT allowed connections

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 6.2.10 and 6.2.10.1).

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 6.2.10 and 6.2.10.1).

Examples

EXAMPLE 1: Explicit definition of a test system interface

```
type component MyMTCType
{
  var integer vc_myLocalInteger;
  timer tc_myLocalTimer;
  port MyMessagePortType pCO1
}

type component MyTestSystemInterface
{
    port MyMessagePortType      pCO1, pCO2;
    port MyProcedurePortType    pCO3
}

// MyTestSystemInterface is the test system interface
testcase TC_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 TC_MyTestcase2 () runs on MyMTCType {
    // map statements are not needed
    // the testcase behaviour
    // …
}
```

10 Declaring constants

TTCN-3 constants are runtime 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.

If functions are used for the initialization of constants, it is strongly advised to adhere to the rules defined in clause 16.1.4. Not following these rules may cause non-deterministic test executions.

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 global constants or component constants of default or 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 clause.

c) Using the dot notation (see clauses 6.2.1.1, 6.2.2.1 and 6.2.5.1) and index notation (see clauses 6.2.3 and 6.2.7) for referencing a field, alternative or element of an **`address`** value, which actual value is **`null`** shall cause an error.

Examples

```
const integer c_myConst1 := 1;
const boolean c_myConst2 := true, c_myConst3 := false;
```

11 Declaring variables

11.0 General

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.

Variables can be declared lazy using the **`@lazy`** modifier.

Alternatively, variables can be declared fuzzy using the **@fuzzy** modifier.

Lazy and fuzzy features are valid only in the scope, where the variables' names are visible. For example, if a fuzzy variable is passed to a formal parameter declared without a modifier, it loses its fuzzy feature inside the called function. Similarly, if it is passed to a lazy formal parameter, it becomes lazy within the called function.

Whenever a lazy or fuzzy variable is assigned, the TE is required to save the lexical environment (the set of directly or indirectly referenced values and templates) valid at the time of the assignment, so that it is possible to resolve the expression at the time of evaluation of the lazy or fuzzy value or template. If the assignment was made on a lower scope than the evaluation, saving the lexical environment extends lifetime of the referenced variables defined on that lower scope.

Example

```
var @fuzzy integer v_fuzzy := 1;
var integer v_var;
var boolean v_condition := true;
if (v_condition) {
    var integer v_local := 0;
    v_fuzzy := v_local;
    v_local := 10;
}
// although v_local is no longer valid at this point, v_fuzzy still evaluates to 10 because
// the lexical environment is available to the fuzzy variable:
v_var := v_fuzzy;
```

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 [ @lazy | @fuzzy ] 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 value variables at other places than the left hand side of assignments, in return statements, or as actual parameters passed to formal parameters shall cause an error.

e) The initialization or assignment of a fuzzy or lazy variable shall not contain function calls of functions with inout or out parameters. The called functions may use other functions with inout or out parameters internally.

f) If lazy or fuzzy value variables are used in deterministic contexts (i.e. during the evaluation of a snapshot or initialization of global non-fuzzy templates), the same restrictions apply to all functions used in the value assigned to the variable as for functions described in clause 16.1.4.

g) The expression assigned to a lazy or fuzzy variable might contain a direct or indirect reference to this variable. Evaluation of such an expression shall cause a dynamic error.

h) Using the dot notation (see clauses 6.2.1.1, 6.2.2.1 and 6.2.5.1) and index notation (see clauses 6.2.3 and 6.2.7) for referencing a field, alternative or element of an **`address`** value, which actual value is **`null`** shall cause an error.

i) The expression shall evaluate to a value, which is at least partially initialized.

Examples

```
var integer v_myVar0;
var integer v_myVar1 := 1;
var boolean v_myVar2 := true, v_myVar3 := false;
var @lazy integer v_myLazyVar1 := v_myVar1+1;
v_myVar1 := 2;
v_myVar1 := v_myLazyVar1; // v_myLazyVar1 evaluates to 2 + 1
v_myLazyVar1 := v_myLazyVar1 + 1;
v_myVar1 := v_myLazyVar1; // causes an error as v_myLazyVar1 references itself
```

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 [ @lazy | @fuzzy ] [ 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 template variables at other places than the left hand side of assignments, in return statements, or as actual parameters passed to formal parameters shall cause an error.

e) Void.

f) 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.

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

h) Restrictions on templates in clause 15 shall apply.

i) The initialization or assignment of a fuzzy or lazy variable shall not contain function calls of functions with inout or out parameters. The called functions may use other functions with inout or out parameters internally.

j) If lazy or fuzzy template variables are used in deterministic contexts (i.e. during the evaluation of a snapshot or initialization of global non-fuzzy templates), the same restrictions apply to all functions used in the template body assigned to the variable as for functions described in clause 16.1.4.

k) Using the dot notation (see clauses 6.2.1.1, 6.2.2.1 and 6.2.5.1) and index notation (see clauses 6.2.3 and 6.2.7) for referencing a field, alternative or element of an **`address`** value, which actual value is **`null`** shall cause an error.

l) The template body at the right-hand side of the assignment symbol shall evaluate to a value or template, which is type compatible with the variable being declared.

m) The template body at the right-hand side of the assignment symbol shall evaluate to an object that is at least partially initialized.

Examples

```
var template integer  v_myVarTemp1 := ?;
var template MyRecord v_myVarTemp2 := { field1 := true, field2 := * },
                      v_myVarTemp3 := { field1 := ?, field2 := v_myVarTemp1 };
var template @fuzzy float v_fuzzTemp1 := rnd(); // evaluated on every usage
var template @fuzzy MyRecord v_fuzzTemp2 := { rnd() < 0.5, float2int(rnd()) };
var template @lazy float LazyTemp1 := v_fuzzTemp1; // evaluates v_fuzzTemp1
var template @lazy MyRecord v_lazyTemp2 :=
    { v_lazyTemp1 < 0.5, float2int(v_fuzzTemp1) }; // evaluates v_lazyTemp1 and v_fuzzTemp1
v_lazyTemp2.field1 := true; // evaluates v_lazyTemp2 and overwrites field1 with true
```

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 t_myTimer1 := 5E-3;
                // declaration of the timer t_myTimer1 with the default value of 5ms

timer t_myTimer2;   // declaration of t_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, timer or default type or of structured types having fields of port, component, timer 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

15.0 General

Templates are used to either transmit a set of distinct values or to test whether a set of received values matches the template specification. Templates can be defined globally 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 template can be declared fuzzy using the **@fuzzy** modifier.

NOTE 1: Using a fuzzy template from a non-fuzzy template causes evaluation of the fuzzy template. Thus, for unparameterized non-fuzzy templates, the result of the used fuzzy templates will stay the same for every usage.

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** or port type.

b) Templates shall not be of a structured type that contains fields of **default** or port type on any level of nesting.

NOTE 2: The **anytype** type does not include the **default** type nor port types (see clause 6.2.6), so that restriction b) does not apply to anytype templates.

c) The expression or template body initializing a template shall evaluate to a value or template, which is type compatible with the template being declared.

d) The expression or template body initializing a template shall evaluate to a value or a template that is at least partially initialized or to a matching mechanism.

e) The body of a fuzzy template shall not contain function calls of functions with inout or out parameters. The called functions may use other functions with inout or out parameters internally.

f) Fuzzy features are valid only in the scope, where the templates' names are visible. For example, if a fuzzy template is passed to a formal template parameter declared without a modifier, it loses its fuzzy feature inside the called function.

Examples

```
type record MyRecord {
  default def
}
type union MyUnion {
  integer choice1,
  MyRecord choice2
}
template MyUnion m_integerChosen := { choice1 := 5 }
  // shall cause an error as the type MyUnion contains MyRecord, which includes
  // a field of default type.

external function fx_garble(charstring p_str) return p_str;
template @fuzzy charstring m_fuzzy := fx_garble("foobar"); // every usage of m_fuzzy re-
                                                          // evaluates the function call
```

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 m_myTemplate:=
{
    field1 := omit,
    field2 := "My string",
    field3 := true
}

// and a corresponding send operation could be
myPCO.send(m_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 mw_myTemplate:=
{
    field1 := ?,
    field2 := pattern "abc*xyz",
    field3 := true
}

// and a corresponding receive operation could be
myPCO.receive(mw_myTemplate);
```

EXAMPLE 3: Template for receiving messages

```
// When used in a receiving operation this template will match any integer value
template integer mw_myTemplate := ?;

// This template will match only the integer values 1, 2 or 3
template integer mw_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 s_template1:=
{
    Par1 := 1,
    Par2 := 2,
    Par3 := 3
}

template RemoteProc s_template2:=
{
    Par1 := 1,
    Par2 := ?,
    Par3 := 3
}

template RemoteProc s_template3:=
{
    Par1 := 1,
    Par2 := ?,
    Par3 := ?
}

template RemoteProc s_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:s_template1);

// Valid invocation since all in and inout parameters have a distinct value
myPCO.call(RemoteProc:s_template2);

// Invalid invocation causing an error
// since the inout parameter Par3 has a matching attribute not a value
myPCO.call(RemoteProc:s_template3);

// Templates never return values. In the case of Par2 and Par3 the values returned by the
// call operation shall 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:s_template1);

// Valid getcall, it will match if Par1 == 1 and Par3 == 3
myPCO.getcall(RemoteProc:s_template2);

// Valid getcall, it will match on Par1 == 1 and Any value of Par3
myPCO.getcall(RemoteProc:s_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:s_template2 value 4);

// Valid getreply, accepting any reply for RemoteProc
myPCO.getreply(RemoteProc:?);

// Valid getreply, also accepting any reply for RemoteProc
myPCO.getcall(RemoteProc:s_template4 value ?);
```

15.3 Global and local templates

TTCN-3 allows defining global templates and local templates.

Syntactical Structure

```
template [ restriction ] [ @fuzzy ] Type TemplateIdentifier  ["(" TemplateFormalParList ")"]
[ modifies TemplateRef ] ":=" TemplateBody
```

NOTE: The optional restriction part is covered by clause 15.8.

Semantic Description

Global templates shall be defined in the module definitions part. Local templates shall be defined in module control, testcases, functions, altsteps or statement blocks. Both global and local templates shall adhere to the scoping rules specified in clause 5.

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.

Both global and local templates are initialized at the place of their declaration. This means, all template fields which are not affected by parameterization shall receive a value or matching mechanism. Template fields affected by parameterization are initialized at the time of template use.

If functions are used for the initialization of module parameters, it is strongly advised to adhere to the rules defined in clause 16.1.4. Not following these rules may cause non-deterministic test executions.

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 mw_myTemplate (integer p_myFormalParam):=
{
    field1 := p_myFormalParam,
    field2 := pattern "abc*xyz",
    field3 := true
}

// could be used as follows
pco1.receive(mw_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 should 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

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] [ @fuzzy ] Type TemplateIdentifier  ["(" TemplateFormalParList ")"]
modifies TemplateRef ":=" TemplateBody
```

NOTE 1: 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 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:

a) In case of templates, template fields or list elements of simple types, **union** and **enumerated** types, the matching mechanism specified in the modified template is simply replacing its corresponding content in its parent.

b) For templates, template fields and elements of **record** and **set** types, if a **record** or **set** field and its corresponding matching mechanism is specified in the modified template, then the specified matching mechanism replaces the one specified in the corresponding field of the parent template. If a **record** or **set** field or its corresponding matching mechanism is – implicitly or explicitly by using the not used symbol "-" - left unspecified in the modified template, then the matching mechanism in the corresponding field of 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).

c) For templates, template fields and elements of **record of** and **set of** types, the above rules specified for **record**s and **set**s apply with the following deviations:

   - if the value list notation is used, only the number of elements listed in the modified template is inherited from the parent (i.e. the list is truncated at the last element of the list notation in the modified template);

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

In case of **record of** and **set of** types first apply rule (c) to the complete structure (e.g. truncation) than apply further rules for the remaining individual type structure elements (see example 3).

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.

A modified template may also be declared fuzzy using the **@fuzzy** modifier.

NOTE 2: If a fuzzy modified template modifies a non-fuzzy unparameterized template, the inherited fields before modification will be the same for every evaluation of the fuzzy template.

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 and change types or names of parameters defined at any of the modification steps between the base template and the actual modified template;

   2) a template parameter restriction of a derived template specified at any of the modification steps between the base template and the actual modified template can be changed to a stricter one (see clause 15.8);

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

   4) 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: Modifying record templates (non-embedded case)

```
// Modifying records
type record MyRecordType
{
    integer field1 optional,
    charstring field2,
    boolean field3
}
template MyRecordType m_myRecTemplate1 :=
{
    field1 := 123,
    field2 := "A string",
    field3 := true
}
// then writing
template MyRecordType m_myRecTemplate2 modifies m_myRecTemplate1 :=
{
    field1 := omit,                 // field1 is optional but present in m_myTemplate1
    field2 := "A modified string"
                                    // field3 is unchanged
}
// is the same as writing
// template MyRecordType m_myRecTemplate2 :=
// {
//    field1 := omit,
//    field2 := "A modified string",
//    field3 := true
// }

template MyRecordType m_myRecTemplate3 modifies m_myRecTemplate1 := {omit, "A modified string"}
//field3 is implicitly left unchanged;
//m_myRecTemplate3 has the same content as m_myRecTemplate2

template MyRecordType m_myRecTemplate4 modifies m_myRecTemplate1 := {omit,"A modified string",-}
//field3 is explicitly left unchanged;
//m_myRecTemplate4 has the same content as m_myRecTemplate2 and m_myRecTemplate3
```

EXAMPLE 2: Modifying record of templates (non-embedded case)

```
type record of integer MyRecordOfType;

template MyRecordOfType m_myBaseTemplate := { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

```
template MyRecordOfType m_myRecOfTemplate1 modifies m_myBaseTemplate :=
                                                    { -, -, 3, 2, -, -, -, -, -, - };
// m_myRecOfTemplate1 contains { 0, 1, 3, 2, 4, 5, 6, 7, 8, 9 }

template MyRecordOfType m_myRecOfTemplate2 modifies m_myBaseTemplate := { -, -, 3, 2 };
// m_myRecOfTemplate2 replaces m_myBaseTemplate with: { 0, 1, 3, 2 };
// elements 5 to 10 of m_myBaseTemplate are truncated

template MyRecordOfType m_myRecOfTemplate3 modifies m_myBaseTemplate := { [2] := 3, [3] := 2 };
// m_myRecOfTemplate3 has the same content as m_myMod1Template: { 0, 1, 3, 2, 4, 5, 6, 7, 8, 9 }
```

EXAMPLE 3: Modifying embedded record and record of templates

```
//Modifying a record embedded in a record of
type record of record {
  integer a,
  integer b
} MyListType

template MyListType mw_myBaseListTemplate := { ?, { a := 1, b := 2 }, ?, { a := 3, b := 4 } }

template MyListType mw_myListTemplate1 modifies mw_myBaseListTemplate := { [1] := { a := 42 } }
//Content of field "a" of the second element is modified,
//the content of mw_myListTemplate1 is: { ?, { a := 42, b := 2 }, ?, { a := 3, b := 4 } }

template MyListType mw_myListTemplate2 modifies mw_myBaseListTemplate := { -, { a := 42 } ,- }
//Content of field "a" of the second element is modified, and the
//record of is truncated after the third element: { ?, { a := 42, b := 2 }, ? }
```

EXAMPLE 4: Modified in-line template

```
// Given
template MyRecordType m_setup :=
{
    field1 := 75,
    field2 := "abc",
    field3 := true
}

// Could be used to define an in-line modified template of Setup
// pco1.send (modifies m_setup := {field1:= 76});
```

EXAMPLE 5: Modified parameterized template

```
// Given
template MyRecordType m_myTemplate1(integer p_myPar):=
{
    field1 := p_myPar,
    field2 := "A string",
    field3 := true
}

// then a modification could be
    template MyRecordType m_myTemplate2(integer p_myPar) modifies m_myRecTemplate1 :=
    // field1 is parameterized in m_myTemplate1 and remains also parameterized in m_myTemplate2
{
    field2 := "A modified string"
}
```

EXAMPLE 6: Default values of modified parameterized templates

```
// Given
template MyRecordType m_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 m_myTemplate12(integer p_int) modifies m_myTemplate11 :=
  // p_int had a default value in m_myTemplate11 but has none in this template
{
    field2 := "B string"
}
```

```
template MyRecordType m_myTemplate13(integer p_int := 0) modifies m_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 m_myTemplate14(integer p_int := - ) modifies m_myTemplate13 :=
  // p_int inherits the default value 0 from its parent m_myTemplate13
{
  field2 := "C string"
}

template MyRecordType m_myTemplate15(integer p_int := - ) modifies m_myTemplate14 :=
  // p_int inherits the default value 0 from m_myTemplate13 via m_myTemplate14
{
  field2 := "D string"
}

template MyRecordType m_myTemplate16(integer p_int) modifies m_myTemplate15 := { }
  // p_int has no default value; no change in the template's content


template MyRecordType m_myTemplate17(integer p_int := - ) modifies m_myTemplate16 :=
  // causes an error as p_int has no default value in the parent template m_myTemplate16
{
  field2 := "E string"
}
```

15.6 Referencing elements of templates or template fields

15.6.0 General

This clause defines rules and restrictions for referencing elements of templates or template fields in case of unrestricted templates or templates with the present restriction. When referencing elements of templates or templates fields with the value or omit restriction, the rules for referencing elements of values are used.

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.4.2) shall be used.

EXAMPLE:

```
var template charstring v_char1 := "MYCHAR";
var template charstring v_char2;

v_char2 := v_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*, template lists and complemented lists: referencing a subfield within a structured field to which *Omit*, *AnyValueOrNone*, a template 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 template lists or complemented template 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 v_r1 := {
    f1 := 5,
    f2 := omit
}
var template R2 v_r2 := v_r1.f2.g2;
    // causes an error as omit is assigned to v_r1.f2
v_r1.f2 := *;
v_r2 := v_r1.f2.g2;
    // causes an error as * is assigned to v_r1.f2

v_r1 := ({f1:=omit, f2:={g1:=0, g2:=omit}},{f1:=5, f2:={g1:=1, g2:={g1:=2, g2:=omit}}});

v_r2 := v_r1.f2;
v_r2 := v_r1.f2.g2;
v_r2 := v_r1.f2.g2.g2;
    // all these assignments cause error as a template list is assigned to v_r1

v_r1 :=
    complement({f1:=omit, f2:={g1:=0, g2:=omit}},{f1:=5, f2:={g1:=1, g2:={g1:=2, g2:=omit}}});

v_r2 := v_r1.f2;
v_r2 := v_r1.f2.g2;
v_r2 := v_r1.f2.g2.g2;
    // all these assignments cause errors as a complemented list is assigned to v_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:

```
v_r1 := {f1:=0, f2:=?}
v_r2 := v_r1.f2.g2;
    // after the assignment v_r2 will be {g1:=?, g2:=*}
v_r1.f2.g2.g2 := ({g1:=1, g2:=omit},{g1:=2, g2:=omit});
    // first the field v_r1.f2 has hypothetically be expanded to {g1:=?,g2:={g1:=?,g2:=*}}
    // thus after the assignment v_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).

d) Special value **`null`**: referencing a field of an **`address`** type, which actual value is **`null`** shall cause an error.

15.6.3 Referencing **`record of`** and **`set of`** elements

Both templates and template variables allow referencing elements of a **`record of`**, array 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*: referencing an element within a record of, set of or array field to which omit is assigned shall follow the rules specified in clause 6.2.3.

b) Template lists, complemented lists, subset and superset: referencing an element within a record of or set of field to which a complemented list, a subset or a superset is assigned, shall cause an error.

EXAMPLE 1:

```
type record of integer RoI;
:
var template RoI v_roI;
var template integer v_int;
v_roI := ({},{0},{0,0},{0,0,0});
v_int := t_RoI[0];
    // shall cause an error as template list is assigned to v_roI
```

c) *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 v_roI;
var template RoRoI v_roRoI;
var template integer v_int;
:
v_roI := ?;
v_int := v_roI[5];
    // after the assignment v_int will be AnyValue(?);

v_roRoI := ?;
v_roI := v_roRoI[5];
    // after the assignment v_roI will be AnyValue(?);
v_int := v_roRoI[5].[3];
    // after the assignment v_int will be AnyValue(?);

v_roI := ? length (2..5);
v_int := v_roI[3];
    // after the assignment v_int will be AnyValue(?);
v_int := v_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);

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

d) *AnyValueOrNone*: referencing an element within a record of, set of or array field to which *AnyValueOrNone* with or without a length attribute is assigned on the right hand side of an assignment shall cause an error. When referencing an element within a record of, set of or array field to which *AnyValueOrNone* is assigned on the left hand side of an assignment, the rules for *AnyValue* shall apply (see item c) for more details).

EXAMPLE 3:

```
type record of integer RoI;
type record R
{
    RoI field1 optional
}
:
var template R mw_t1 := { field1 := * };
var template integer mw_t2;
mw_t1.field1[2] := 2; // after the assignment, mw_t1 will be { field1 := { ?, ?, 2, * } }
mw_t1.field1 := *;
mw_t2 := mw_t1.field1[0];
    // shall cause an error as mw_t1.field1 contains AnyValueOrNone
```

e) *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. *AnyElementsOrNone* as a permutation element causes that the permutation shelters all record of element indexes.

EXAMPLE 4:

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

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

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

f) 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).

g) *AnyElementsOrNone*: when referencing an element of a record of or set of template or field that contains *AnyElementsOrNone*, the result of an operation is dependent on the position of *AnyElementsOrNone*, the referenced index and length attributes attached to *AnyElementsOrNone*.

When resolving the reference, a transformed form of the record of or set of template is used. The transformed form is equal to the original value where all occurrences of *AnyElementsOrNone* with a length restriction are replaced with a sequence of AnyElements of the same size as the lower bound. If the lower bound is greater than the upper bound, the sequence shall be followed by a single *AnyElementsOrNone* symbol with a length restriction. The lower bound of this restriction is zero and the upper bound is the difference between the lower and upper bound of the original restriction.

EXAMPLE 5:

```
type record of interger RoI;
template RoI mw_roI := {1, * length(2), 5};     // transformed form: {1, ?, ?, 5}
template RoI mw_roI := {1, * length(1..3), 5};  // transformed form: {1, ?, * length(0..2), 5}
```

h) Special value **null**: referencing an element of an **address** type, which actual value is **null** shall cause an error.

When the reference is used at the right hand side of the assignment, the following applies:

- If positions of all *AnyElementsOrNone* matching symbols in the transformed form are greater than the position of the referenced item, rules from the clause 6.2.3.2 are used for resolving the reference.

EXAMPLE 6:

```
type record of interger RoI;
:
var template RoI v_roI := {1, 2, * length(2), 5};
    // transformed form: {1, 2, ?, ?, 5}
var template integer v_int;
v_int := v_roI[1]; // after the assignment, v_int will be 2
v_int := v_roI[2]; // after the assignment, v_int will be ?
```

- If the position of the referenced item is greater or equal to the position of any *AnyElementsOrNone* symbol in the transformed template, an error is generated.

EXAMPLE 7:

```
type record of interger RoI;
:
var template RoI v_roI := {1, 2, *, 5};
var template integer v_int := v_roI[3]; // produces an error

v_roI := {1, 2, *};
v_int := v_roI[2];  // produces an error
```

When the reference is used at the left hand side of the assignment, the following applies:

- If positions of all *AnyElementsOrNone* matching symbols in the transformed form are greater than the position of the referenced item the following rules are used. If the referenced item is not a result of transformation, the value or matching symbol at the right hand side of the assignment shall replace the referenced symbol in the original template. If the referenced element was a result of transformation, then the *AnyValueOrNone* symbol in the original template is replaced with its transformed form and the assignment is performed afterwards.

EXAMPLE 8:

```
type record of interger RoI;
:
var template RoI v_roI := {1, 2, * length(2), 5};
    // transformed form: {1, 2, ?, ?, 5}
v_roI [1] := 10;    // after the assignment, t_RoI will be {1, 10, * length(2), 5}
v_roI [2] := 3;     // after the assignment, t_RoI will be {1, 10, 3, ?, 5}
```

- If the position of the referenced item is greater or equal to the position of any *AnyElementsOrNone* symbol in the transformed form and this *AnyElementsOrNone* symbol is not the last element in the template, an error is generated.

EXAMPLE 9:

```
type record of interger RoI;
:
var template RoI v_roI:= {1, 2, *, 5};
v_roI[3] := 4; // produces an error
```

- If the position of the referenced item is greater or equal to the position of an *AnyElementsOrNone* symbol in the transformed form and this *AnyElementsOrNone* is the last symbol in the template, the value or matching symbol at the right hand side of the assignment shall be assigned to the referenced element. Then the *AnyElementsOrNone* symbol and all unbound values between it and the referenced symbol shall be replaced with *AnyElement* symbols. If the *AnyElementsOrNone* symbol had a length restriction, only as many *AnyElement* symbols can be added as is the value of the upper bound of the restriction. As the last step, an *AnyElementsOrNone* symbol can be appended to the end of the template. The symbol is always appended if the original *AnyElementsOrNone* symbol was unrestricted. If the original *AnyElementsOrNone* had a length restriction, the symbol is appended only if the restriction included items beyond the referenced item. In such a case, the appended symbol contains the original length restriction adjusted by the difference between the size of the template before and after assignment.

EXAMPLE 10:

```
type record of interger RoI;
:
var template RoI v_roI := {1, 2, * };
v_roI[4] := 5; // {1, 2, ?, ?, 5, *};

v_roI := {1, * length(1..2)};
v_roI[4] := 5; // {1, ?, ?, -, 5};
    // short length restriction: only two ? symbols added and no * at the end

v_roI := {1, * length(1..5)};
v_roI[2] := 3; // {1, ?, 3, * length(0..3)};
    // adjusted length restriction at the end
```

The index of the referenced item shall not violate type restrictions in any of the above cases.

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 s_mySign1 := ({ par1 := 1, par2 := 2 }, { par1 := 2, par2 := 1 });
template MySignature s_mySign2 modifies s_mySign1 := { par1 := ? };
// shall cause an error as s_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 s_mySign3 := ?;
template MySignature s_mySign4 modifies s_mySign3 := { par1 := 3 };
// s_mySign3 is expanded to { par1 := ?, par2 := ? }, then 3 is assigned to par1,
// thus s_mySign4 will be { par1 := 3, par2 := ? }
```

15.6.5 Referencing **union** alternatives

Both templates and template variables allow referencing alternatives inside a union template definition using the dot notation. However, the referenced alternative may belong to template field containing a matching mechanism. This clause provides rules for such cases.

a) *Omit*, *AnyValueOrNone*, template lists and complemented lists: referencing an alternative of a union template or template field to which *Omit*, *AnyValueOrNone*, a template list or a complemented list is assigned, at the right hand side of an assignment, shall cause an error.
When referencing an alternative of a union template or template field to which *AnyValueOrNone* or omit is assigned, at the left hand side of an assignment, the template field is implicitly set to be present and the referenced alternative becomes the chosen one. If the referenced alternative is not the last element of the dot notation, rules in clause 15.6.2 valid for *AnyValue* shall apply recursively for further expansion. After this expansion the value or matching mechanism at the right hand side of the assignment shall be assigned to the referenced subfield.
Referencing an alternative of a union template field to which template lists or complemented template lists are assigned, at the left hand side of an assignment, shall cause an error.

EXAMPLE 1:

```
type record R1 {
    integer f1,
    integer f2
}
type union U {
    integer c1,
    R1      c2
}
type record R2 {
    integer g1,
    U       g2 optional
}
:
var template R2 v_t1 := {
    g1 := 5,
    g2 := *
}
var template integer v_t2;
v_t1.g2.f1 := 1;
    // after the assignment v_t2.g2 is { g2 := { f1 := 1, f2 := ? } }
v_t1.g2 := omit;
v_t2 := v_t1.g2.c1;
    // causes an error as omit is assigned to v_t1.g2
```

b) *AnyValue*: when referencing an alternative of a union template or template field to which *AnyValue* is assigned, at the right hand side of an assignment, *AnyValue* shall be returned.
When referencing an alternative of a union template or template field to which *AnyValue* is assigned, at the left hand side of an assignment, the referenced alternative becomes the chosen one. If the referenced alternative is not the last element of the dot notation, rules in clause 15.6.2 valid for *AnyValue* shall apply recursively for further expansion. 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:

```
var template U v_t3 := ?;
v_t2 := v_t3.c1;
    // after the assignment v_t2 will be ?
v_t3.c1.f1 := 1;
    // after the assignment v_t3 will be { c1 := { f1 := 1, f2 := ? } }
```

c) Ifpresent attribute: referencing an alternative of a union template 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).

d) Special value **`null`**: referencing an alternative of an **`address`** type, which actual value is **`null`** shall cause an error.

15.7 Template matching mechanisms

15.7.0 General

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.

Restrictions

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

a) All other applications of matching mechanisms than the ones allowed in table 11 are forbidden.

Table 11: TTCN-3 Matching Mechanisms

Used with values of	Value	Instead of values										Inside values			Attributes	
	SpecificValue	Omit	ComplementedList	TemplateList	AnyValue (?)	AnyValueOrNone (*)	Range	Superset	Subset	Pattern	Match decoded content	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	Yes[1]
octetstring	Yes	Yes[1]	Yes	Yes	Yes	Yes[1]					Yes	Yes	Yes		Yes	Yes[1]
hexstring	Yes	Yes[1]	Yes	Yes	Yes	Yes[1]					Yes	Yes	Yes		Yes	Yes[1]
character strings	Yes	Yes[1]	Yes	Yes	Yes	Yes[1]	Yes			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	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 of any type as a whole or to optional fields of record and set templates. However when matching, it 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 | all from TemplateInstance) ["," ] } ")" |
complement "(" { (TemplateInstance | all from TemplateInstance) ["," ] } ")" |
"?" |
"*" |
"(" ( ConstantExpression | -infinity ) ".." ( ConstantExpression | infinity ) ")" |
superset "(" { (TemplateInstance | all from TemplateInstance) ["," ] } ")" |
subset "(" { (TemplateInstance | all from TemplateInstance) ["," ] } ")" |
pattern [@nocase] Cstring
decmatch [ "(" Expression ]")" ] TemplateInstance
EnumValueIdentifier "(" TemplateBody {"," TemplateBody} ")"
```

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;

NOTE 1: **omit** can be assigned to templates of any type as a whole or to optional fields of record and set types. **omit** can only be used for matching optional fields.

- **(…):** 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;

NOTE 2: ***** can be assigned to templates of any type as a whole or to optional fields of record and set types. ***** can only be used for matching optional fields.

- **(***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;
- **decmatch:** used for matching of encoded payload fields;
- **EnumValueIdentifier with list of templates:** used for matching of enumerated values with associated value list.

The matching mechanisms list, complemented list, subset, and superset can use the elements of a template using the **all from** clause.

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.

b) All templates and values used in the matching mechanisms above (including the referenced ones, e.g. within a pattern) shall be completely initialized.

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 | "?" | "*" | all from TemplateInstance )[","] } ")" …
```

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 and all elements of a template can be added to permutation using the **all from** clause).

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.

b) All templates or values used in the permutation matching mechanism shall be completely initialized.

Examples

```
template bitstring mw_b := '10???'B;        // where each "?" may either be 0 or 1
type record of integer RI;
template RI mw_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).

NOTE 1: **ifpresent** can be assigned to templates of any type as a whole or to optional fields of record and set types. **ifpresent** can only be used for matching optional fields.

NOTE 2: Assigning **ifpresent** to a template that already matches the special value omit (i.e. it is either **omit**, an **ifpresent** template or *AnyValueOrNone*) has no effect; the resulting template will match the same set of values and the special value **omit** as the template the **ifpresent** is assigned to.

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.

b) All values used in the length matching attribute shall be completely initialized.

Examples

```
type record R {
    record of integer ri optional
}
template R mw_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	Omit	ComplementedList	TemplateList	AnyValue (?)	AnyValueOrNone (*)	Range	Superset	Subset	Pattern	Match decoded content	AnyElement (?)	AnyElementsOrNone (*)	Permutation	LengthRestriction	IfPresent
omit	Yes	Yes														
value	Yes	Note 1														
present	Yes	Note 1	Yes	Yes	Yes	Note 1	Yes	Yes	Yes	Yes	Note 2	Yes	Yes	Yes	Yes	Note 1

NOTE 1: 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.
NOTE 2: The matching mechanism is allowed only if the template following the **decmatch** keyword is fulfilling the given restriction.

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

Formal parameter/-left hand side of an expression \ Actual parameter/right hand side of an expression	value	template (omit)	template (value)	template (present)	template
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.					

Examples

```
// definitions of restricted templates
type record ExampleType {
    integer a,
    boolean b optional
}

template(omit) ExampleType m_exampleOmit := omit;
template(omit) ExampleType m_exampleOmitValue:= { 1, true };
template(omit) ExampleType mw_exampleOmitAny := ?;           // incorrect

template(value) ExampleType m_exampleValueomit := omit;     // incorrect
template(value) ExampleType m_exampleValue := { 1, true };
template(value) ExampleType m_exampleValueOptional := { 1, omit };
                                                    // omit assigned to a field is correct

template(present) ExampleType mw_examplePresent := {1, ?};
template(present) ExampleType mw_examplePresentIfpresent := { 1, true } ifpresent;
                                                    // incorrect
template(present) ExampleType mw_examplePresentAny := ?;


// restricted template usage
var template (omit) ExampleType v_omit;
var template (present) ExampleType v_present;
var template (value) ExampleType v_value;

v_omit := m_exampleOmit;
v_omit := m_exampleValueOptional;
v_omit := mw_examplePresentAny;     // incorrect, not a specific value

v_present := m_exampleOmit;         // incorrect, shall not be omit
v_present := mw_examplePresent;

v_value := m_exampleOmit;           // incorrect, shall not be omit
v_value := mw_examplePresentAny;    // incorrect, shall be a single value
```

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. It matches an expression, which shall denote a value or a field of a value against a template instance. Types of the expression and the template instance shall be compatible (see clause 6.3). The return value of the **match** operation indicates whether the expression matches the specified template instance. In the special case, matching a non-optional value expression (e.g. a value variable or non-optional field of a value) with a template instance that matches an omitted field (i.e. one of the matching mechanisms *Omit*, *AnyValueOrNone*, *IfPresent*) shall be allowed and shall be treated as if the value expression were an optional field. Thus, matching a value expression against a template instance which evaluates to the **omit** matching mechanism shall return **false**.

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 evaluate to a value or shall denote an omitted optional field, i.e. the **match** operation cannot be used to compare two templates.

b) The operands of the **match** operation shall be completely initialized.

c) The type of the template instance-parameter shall be unambiguously identified. If the expression-parameter evaluates to a literal value without explicit or implicit identification of its type, the type of the template instance-parameter shall be used as the type governor for the expression-parameter.

NOTE: In case of in-line templates, see restriction b) in clause 15.4.

Examples

EXAMPLE 1: Using the match operation

```
template integer mw_lessThan10 := (-infinity..9);
:
myPort.receive(integer:?) -> value v_rxValue;
if( match( v_rxValue, mw_lessThan10)) { … }
// true if the actual value of v_rxvalue is less than 10 and false otherwise
:

type record R { integer a, integer b optional, integer c optional }
const R c_r := { a := 1, b := omit, c := 1 }
const integer c_c := 1;
:
function f_f(template(omit) integer p_o) {
    :
    match(c_c, omit)    // returns false
    match(5, omit)      // returns false
    match(c_c, *)       // returns true
    match(c_r, c_c)     // error (different types)
    match(c_r.a, p_o)   // returns true if p_o evaluates to 1, false, otherwise
    match(c_r.b, p_o)   // returns true, if p_o is not present, false, otherwise
    match(c_r.c, p_o)   // returns true, if p_o evaluates to 1, false, otherwise
    match(c_c, p_o)     // returns true, if p_o evaluates to 1, false, otherwise
    match(c_c, 1)       // returns true (the syntax of the template parameter implicitly
                        // identifies its type, see clause 15.4)
}
```

EXAMPLE 2: Using the match operation with enumerated types

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

type enumerated MySecondEnumType { Saturday, Sunday, Monday };

control {
    var MyFirstEnumType v_today := Tuesday;
    match (v_today, Sunday) // causes an error, as the value Sunday alone does not specifies
    // the type context of the template instance-parameter
    match (v_today, MySecondEnumType:Sunday) // returns false
    match (Monday, v_today)
    //returns false; in this case v_today is governing the type context for the match operation
    //(MyFirstEnumType), but its actual value is different from Monday
}
```

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 m_setupTemplate :=
{
    field1 := 1,
    field2 := true
}

:
var ExampleType v_rxValue := valueof(m_setupTemplate);
```

EXAMPLE 2:

```
function MyFunc() {
  var template integer v_tInt := omit;
  //is ok, but to be used for optional record or set fields only
  var integer v_int := valueof(v_tInt)
  //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 (in-line) templates using the concatenation operation. With the exception of charstring and universal charstring templates, 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* without a length modifier, *AnyValue* or *AnyValueOrNone*, both constrained to a fixed length, *AnyElement* or *AnyElementsOrNone* possibly constrained with a length attribute for list types. The length matching attribute shall not follow a template or template field produced by concatenation directly, but in this case the concatenation shall be placed within a pair of parentheses.

Single templates of charstring and universal charstring types shall contain specific values only. When concatenating templates of charstring and universal charstring types, each single template shall be either of the charstring or universal charstring type. When templates of charstring and universal charstring type are both present in the concatenation, the charstring values are implicitly converted to universal charstring values according to the rules specified in clause 6.3.1 before concatenation and the resulting template is of the universal charstring type.

The concatenation results in the sequential concatenation of the single templates from left to right, with two exceptions: matching symbol *AnyValue* without a length modifier shall be replaced by a single *AnyElementsOrNone* matching symbol before concatenation and matching symbols *AnyValue* and *AnyValueOrNone* that are each 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: See also concatenation of character string patterns in clause B.1.5.

Restrictions

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

a) All operands of the concatenation operation shall be at least partially initialized.

EXAMPLE 1: Composing templates of string types

```
template charstring mw_mychar1 := "ABC" & "DE*" & "F?";
    // results in the template "ABCDE*F?"
    // please note that "*" and "?" denote the characters "*" and "?"

template charstring mw_mychar2 := "ABC" & * length(2) & "EF";
    // causes an error as for character string types only
    // specific values are allowed

template bitstring mw_mybit := '010'B & ? & '1'B & ? length(1) & '1'B;
    // results in the template '010*1?1'B
    // note that & ? & turns to * within the resulting bitstring as the original ?
    // stands for a bitstring of any length

template octetstring mw_myoct1 := 'ABCD'O & 'EF'O & ? & ? length(1) & 'EF'O;
    // results in the template 'ABCDEF*?EF'O
    // note that & ? & turns to * within the resulting octetstring as the original ?
    // stands for an octetstring of any length

template octetstring mw_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 mw_myoctWrong := 'ABCD'O & ? length(2) length (4);
    // causes an error, no length matching attribute shall directly follow a concatenation

template octetstring mw_myoct3 := ('ABCD'O & ? length(2)) length (1..3);
    // However, this is correct but will not match any value;

template hexstring mw_myhexPar (integer N):=
    'ABC'H & ? length(N) & 'E'H & ? length(1) & 'F'H;
function f_myFunc() runs on MyCompType {
  var integer v_int := 3;
  var template hexstring v_hstring;
  :
  v_hstring := 'ABC'H & ? length(v_int) & 'E'H & ? length(1) & 'F'H;
    //results in the template 'ABC???E?F'H
  p.receive (mw_myhexPar(4));
    //actual content of mw_myhexPar is 'ABC????E?F'H
}
```

EXAMPLE 2: Composing templates of list types

```
type record of charstring RecofChar;
type set of integer SetofInt;

template RecofChar mw_myRecofChar := {"ABC"} & {"D?", "EF"};
    // results in the template {"ABC", "D?", "EF" }

template SetofInt mw_mySetofInt := { 1, 2 } & ? length(2) & { 3, 4 };
    // results in the template {1, 2, ?, ?, 3, 4 }

template RecofInt mw_myRecofInt := { 1, 2 } & { * length(2), 3, 4 };
    // results in the template {1, 2, ?, ?, 3, 4 }

template RecofChar mw_myRecofCharWrong:= {"ABC"} & ? length(1..2) & {"EF"};
    // causes an error, the length attribute shall denote a fixed length
```

```
template RecofChar mw_myRecofCharPar (integer N):= { "ABC" } & ? & * length(N) & { "EF" };
function MyFunc() runs on MyCompType{
  var integer v_int := 3;
  var template RecofChar v_recofChar;
  :
  v_recofChar :=  { "ABC" } & ? length(v_int) & { "EF" };
    //results in the template { "ABC", ?, ?, ?, "EF" }
  p.receive ( mw_myRecofCharPar(3) );
    //actual content of mw_myRecofCharPar is { "ABC", ?, ?, ?, ?, "EF" }
}
```

16 Functions, altsteps and testcases

16.0 General

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

16.1.0 General

Functions are used in TTCN-3 to express test behaviour, to organize test execution or to structure computation in a module, for example, to calculate a single value, to initialize a set of variables or to check some condition.

Syntactical Structure

```
function [ @deterministic ]  FunctionIdentifier
"(" [ { ( FormalValuePar | FormalTimerPar | FormalTemplatePar | FormalPortPar ) [","] } ] ")"
[ runs on ComponentType ]
[ mtc ComponentType ]
[ system 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 evaluation of the return value or template, 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 26.

Functions may be parameterized.

Functions may have an **mtc** clause. If a function has an **mtc** clause, the type referenced by this clause shall be mtc-compatible (see clause 6.3.3) with the type of the **mtc** component reference. If the mtc clause is not present, the type of the **mtc** component reference is unknown in the scope of this function.

Functions may have a **system** clause. If a function has a **system** clause, the type referenced by this clause shall be system-compatible (see clause 6.3.3) with the type of the **system** component reference**.** If the system clause is not present, the type of the **system** component reference is unknown in the scope of this function.

Using the **`@deterministic`** modifier, a function can be declared to be deterministic. Deterministic functions are safe to be used when called from specific places where non-determinism could lead to unexpected side effects (see clause 16.1.4).

NOTE 0: The determination of determinism of a function is a semi-decidable problem and as such can and will not be exhaustively checked. As such, the annotation deterministic is mainly used for informational purposes and for allowing certain functions to be used during snapshot evaluation. Principally, a function can be seen as deterministic if it does not violate any of the restrictions from clause 16.1.4 which does not mean that violation of these restriction automatically leads to non-determinism.

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 within behaviours 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`**, **`mtc`** or **`system`** 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 reference (including calling a value or template returning function)or template instance compatible to the type specified in the function header return clause. If the **`return`** clause has a template restriction, this restriction shall be adhered to by the returned template. The return statement shall return a template that is at least partially initialized.

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.

j) If a function references the names of definitions that are defined inside 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.

Examples

EXAMPLE 1: Function with return

```
// Definition of f_myFunction which has no parameters
function f_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 f_myFunction2() return template integer
{
:
    return ?;   // returns the matching mechanism AnyValue
}
function f_myFunction3() return template octetstring
{
:
```

```
    return 'FF??FF'O;   // returns an octetstring with AnyValue inside it
}
```

EXAMPLE 3: Function with runs on clause

```
function f_myFunction3() runs on MyPTCType {
                                // f_myFunction3 does not return a value, but
    var integer v_myVar := 5;   // does make use of the port operation
    pCO1.send(v_myVar);         // send and therefore requires a runs on
                                // clause to resolve the port identifiers
}                               // by referencing a component type
```

EXAMPLE 4: Parameterized function

```
function f_myFunction2(inout integer p_myPar1) {
                                // f_myFunction2 does not return a value
    p_myPar1 := 10 * p_myPar1;  // but changes the value of p_myPar1 which
}                               // is passed in by reference
```

EXAMPLE 5: Function without return statement

```
function f_myFunction5(inout integer p_myPar1) return integer {
    if (p_myPar1 > 5) {
      p_myPar1 := 5;
      return p_myPar1;
    }
    // in case of p_myPar1 <= 5, f_myFunction5 does not terminate in a return statement
    // and will cause a test case error
}
```

EXAMPLE 6: Function with system and mtc

```
type component MtcType { ... }
type component SystemType { ... }

function f_myFunction6() runs on MyPtcType mtc MtcType system SystemType {
    var MtcType v_mtc := mtc;
    var SystemType v_system := system;
    f_myFunction3(); // allowed, f_myFunction3() has no mtc and system clause
    f_myFunction6(); // allowed, f_myFunction6() has compatible mtc and system clause
}
function f_myFunction7() runs on MyPtcType system SystemType {
    var MtcType v_mtc := mtc; // not allowed, mtc type unknown
    f_myFunction6(); // possible runtime error, no mtc clause of f_myFunction7
}
function MyFunction8() runs on MyPtcType mtc MtcType {
    var SystemType v_system := system; // not allowed, system type unknown
    f_myFunction6(); // possible runtime error, no system clause of f_myFunction8
}
```

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.

f) When invoking a function, the mtc and system compatibility of the mtc and system components of the invoked function with the actual mtc and system types of the running test case as described in clause 6.3.3 need to be fulfilled.

Examples

```
v_myVar := f_myFunction4(); // The value returned by f_myFunction4 is assigned to v_myVar.
                            // The types of the returned value and v_myVar have to be compatible

f_myFunction2(v_myVar2);    // f_myFunction2 does not return a value and is called with the
                            // actual parameter v_myVar2, which may be passed in by reference

v_myVar3 := f_myFunction6(4) + f_myFunction7(v_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 14.

Table 14: 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 enumerated value	**`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 **octetstring** value	**`unichar2oct`**
	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 **octetstring** value to **universal charstring** value	**`oct2unichar`**
	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 enumerated value to **integer** value	**`enum2int`**
	Convert value or template to **universal charstring** value	**`any2unistr`**
Length/size functions	Return the length of a value or template of any string type, **record of**, **set of** or **array**	**`lengthof`**

Category	Function	Keyword
	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 or is assigned a matching mechanism that cannot match an ommitted field (i.e. none of **omit**, *AnyValueOrNone* or **ifpresent**)	**ispresent**
	Determine which choice has been selected in a **union** value or template	**ischosen**
	Determine if a template evaluates to a concrete value	**isvalue**
	Determine if a template is uninitialized or not	**isbound**
	Determine if a template contains certain matching mechanism	**istemplatekind**
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**
	Encode a value into a universal charstring	**encvalue_unichar**
	Decode a universal charstring into a value	**decvalue_unichar**
	Encode a value into a octetstring	**encvalue_o**
	Decode a octetstring into a value	**decvalue_o**
	Retrieve the type of string encoding	**get_stringencoding**
	Remove BOMs of UCS encoding schemes	**remove_bom**
Other functions	Generate a random float number	**rnd**
	Returns the name of the currently executing test case	**testcasename**
	Returns the host id of the test component or module	**hostid**

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 ")" |
unichar2oct "(" SingleExpression ["," 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 ")" |
oct2unichar "(" SingleExpression ["," SingleExpression] ")" |
str2int "(" SingleExpression ")" |
str2hex "(" SingleExpression ")" |
str2oct "(" SingleExpression ")" |
str2float "(" SingleExpression ")" |
enum2int "(" SingleExpression ")" |
any2unistr "(" SingleExpression ")" |
lengthof "(" TemplateInstance ")" |
sizeof "(" TemplateInstance ")" |
ispresent "(" TemplateInstance ")" |
ischosen "(" TemplateInstance ")" |
isvalue "(" TemplateInstance ")" |
```

```
isbound "(" TemplateInstance ")" |
istemplatekind "(" TemplateInstance "," TemplateInstance ")" |
regexp [@nocase] "(" TemplateInstance"," TemplateInstance"," SingleExpression ")" |
substr "(" TemplateInstance "," SingleExpression "," SingleExpression ")" |
replace "(" SingleExpression "," SingleExpression "," SingleExpression "," SingleExpression ")" |
encvalue "(" TemplateInstance ["," SingleExpression] ["," SingleExpression] ")" |
decvalue "(" SingleExpression "," SingleExpression
            ["," SingleExpression] ["," SingleExpression] ["," SingleExpression] ")" |
encvalue_unichar "(" TemplateInstance ["," SingleExpression]
                    ["," SingleExpression] ["," SingleExpression] ")" |
decvalue_unichar "(" SingleExpression "," SingleExpression
                    ["," SingleExpression] ["," SingleExpression] ["," SingleExpression] ")" |
encvalue_o "(" TemplateInstance ["," SingleExpression] ")" |
decvalue_o "(" SingleExpression "," SingleExpression ["," SingleExpression] ")" |
get_stringencoding "(" SingleExpression ")" |
remove_bom(" SingleExpression ") |
rnd "(" [ SingleExpression ] ")" |
testcasename "()" |
hostid "(" [ SingleExpression ] ")"
```

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 `in` and `inout` parameters shall be initialized with the following exceptions:

      - the actual `in` parameter passed to the predefined functions `isvalue`, `ischosen`, `ispresent` and `isbound` may be uninitialized or even contain non-evaluable reference expressions;
      - `any_string_or_sequence_type` parameters of the functions `lengthof`, `substr` and `replace` may be partially initialized;
      - the `invalue` parameter of the `any2unistr` function may be uninitialized or partially initialized;
      - the `encoded_value` parameter of the `decvalue` and `decvalue_unichar` function may be uninitialized.

b) Restrictions on invoking functions from specific places are described in clause 16.1.4.

Examples

```
var hexstring v_h:= bit2hex ('111010111'B);
var octetstring v_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 [ @deterministic ] ExtFunctionIdentifier
"(" [ { ( FormalValuePar | FormalTimerPar | FormalTemplatePar | FormalPortPar ) [","] } ] ")"
[ return  [ template [ Restriction ] ] 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.

Using the **@deterministic** modifier, an external function can be declared to be deterministic. Deterministic functions are safe to be used when called from specific places where non-determinism could lead to unexpected side effects (see clause 16.1.4).

Restrictions

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

a) Restrictions on invoking functions from specific places are described in clause 16.1.4.

NOTE: External functions should only exchange information with the test system via return values and parameter passing. Side-effects that change the status of the test system and may influence the test outcome should be avoided. Such side-effects can occur if an external function contains default handling, configuration, communication or timer operations.

Examples

```
external function fx_myFunction4() return integer;  // External function without parameters
                                                    // which returns an integer value

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

16.1.4 Invoking functions from specific places

If value returning functions are called in receiving communication operations (in templates, template fields, in-line templates, or as actual parameters), in guards or events of alt statements or altsteps (see clause 20.2), or in initializations of altstep local definitions (see clause 16.2), the following operations shall not be used in functions called in the cases specified above, in order 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:

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**, **checkstate**, **send**, **receive**, **trigger**, **call**, **getcall**, **reply**, **getreply**, **raise**, **catch**, **check**, **connect**, **disconnect**, **map** and **unmap** (see notes 1, 2, 3, 4 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 non-deterministic external functions, i.e. external functions where the resulting values for actual inout or out parameters or the return value may differ for different invocations with the same actual in and inout parameters (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 and deterministic external functions with **out** or **inout** parameters (see notes 7 and 8).

k) Calling functions and external functions with **@fuzzy** formal parameters and variables (see notes 4 and 9).

l) The **setencode** operation (see note 8 and clause 27.9).

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** shall cause 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 non-deterministic external functions, **rnd**, **running**, **alive**, **read**, **checkstate**, **setverdict**, referencing fuzzy objects and writing to component variables causes an error because this 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 in restriction j) 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 and deterministic external 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 and the **setencode** operation causes an error because this may lead to different results of subsequent evaluations of the same snapshot.

NOTE 9: Calling functions and external functions with **@fuzzy** parameters causes an error, because fuzzy objects are re-evaluated each time referenced and this may lead to different results of subsequent evaluations of the same snapshot.

16.2 Altsteps

16.2.0 General

TTCN-3 uses altsteps to specify default behaviour or to structure the alternatives of an **alt** statement.

Syntactical Structure

```
altstep AltstepIdentifier
"(" [ { ( FormalValuePar | FormalTimerPar | FormalTemplatePar | FormalPortPar ) [","] } ] ")"
[ runs on ComponentType ]
[ mtc ComponentType ]
[ system 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.

Altsteps may have an **mtc** clause. If an altstep has an **mtc** clause, the type referenced by this clause shall be mtc-compatible (see clause 6.3.3) with the type of the **mtc** component reference. If the **mtc** clause is not present, the type of the **mtc** component reference is unknown in the scope of this altstep.

Altsteps may have a **system** clause. If an altstep has a **system** clause, the type referenced by this clause shall by system-compatible (see clause 6.3.3) with the type of the **system** component reference**.** If the **system** clause is not present, the type of the **system** component reference is unknown in the scope of this altstep.

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 evaluation of formal parameters' default values and 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 formal parameters' default values and 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.

f) Altsteps 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 within behaviours without a runs on clause.

Examples

EXAMPLE 1: Parameterized altstep with runs on clause

```
// Given
type component MyComponentType {
    var integer vc_myIntVar := 0;
    timer tc_myTimer;
    port myPortTypeOne pCO1, pCO2;
    port myPortTypeTwo pCO3;
}

// Altstep definition using pCO1, pCO2, vc_myIntVar and tc_myTimer of MyComponentType
altstep a_altSet_A(in integer p_myPar1) runs on MyComponentType {
    [] pCO1.receive(mw_myTemplate(p_myPar1, vc_myIntVar)) {
           setverdict(inconc);
       }
    [] pCO2.receive {
           if (p_myPar1 != 0) {
               repeat
           }
           else {
               break
           }
       }
    [] tc_myTimer.timeout {
        setverdict(fail);
           stop
       }
}
```

EXAMPLE 2: Altstep with local definitions

```
altstep a_anotherAltStep(in integer p_myPar1) runs on MyComponentType {
    var integer v_myLocalVar := f_myFunction();         // local variable
    const float c_myFloat := 3.41;                      // local constant
    [] pCO1.receive(MyTemplate(p_myPar1, v_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 20.5.3) 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 1: 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**.

NOTE 2: Due to the possibility of defining dynamic test configurations, an alternative in an explicitly invoked altstep may refer to a disconnected or unmapped port at the time of its evaluation. In TTCN-3, ports belong to the receiving component and matching is related to the top elements in the port queues. Dynamically unmapped and disconnected ports contribute to a snapshot in the same manner as mapped and connected ports. This means, an explicitly invoked **altstep** may execute receiving operations that empty the queues of unmapped and disconnected ports without causing a test case error.

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.

c) When invoking an altstep, the mtc and system compatibility of the mtc and system components of the invoked altstep with the actual mtc and system types of the running test case as described in clause 6.3.3 need to be fulfilled.

Examples

EXAMPLE 1: Implicit invocation of an altstep via a default activation

```
 :
var default v_myDefVarTwo := activate(a_mySecondAltStep()); // Activation of an altstep as
                                                            // default
 :
```

EXAMPLE 2: Explicit invocation of an altstep within an alt statement

```
 :
alt {
    [] pCO3.receive {
         …
       }
    [] a_anotherAltStep();  // explicit call of altstep a_anotherAltStep as an alternative
                            // of an alt statement
    [] t_myTimer.timeout {}
}
```

EXAMPLE 3: Explicit, stand-alone invocation of an altstep

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

//is a shorthand for

alt {
    [] a_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 TC_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 TC_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.

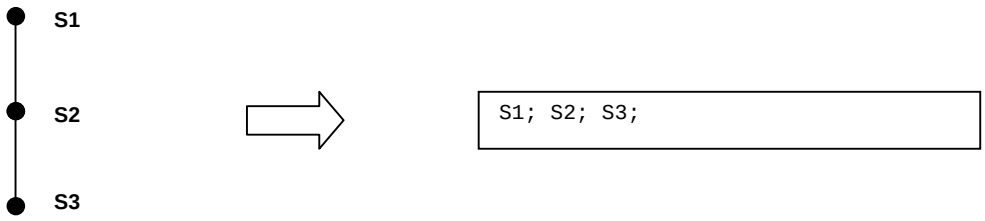


Figure 8: Illustration of sequential behaviour

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 15 gives an overview of the TTCN-3 expressions, statements and operations and restrictions on their usage.

Table 15: Overview of TTCN-3 expressions, statements and operations

Statement	Associated keyword or symbol	Can be directly or indirectly invoked by module control, but not by test components	Can be invoked by functions, test cases and altsteps running on test components	Can be directly or indirectly invoked from specific places (see note 1)
Expressions	**(…)**	Yes	Yes	Yes
Basic program statements				
Assignments	**:=**	Yes	Yes	Yes (see note 4)
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 5)	Yes
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 2)	Yes	
Re-evaluation of alternative behaviour	**repeat**	Yes	Yes	
Interleaved behaviour	**interleave {…}**	Yes (see note 2)	Yes	
Activate a default	**activate**	Yes	Yes	
Deactivate a default	**deactivate**	Yes	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	
Terminating the testcase with an error verdict	**testcase.stop**		Yes	Yes
Remove a test component from the system	**kill**		Yes	
Check termination of a PTC behaviour	**running**		Yes	
Check if a PTC exists in the test system	**alive**		Yes	
Wait for termination of a PTC behaviour	**done**		Yes	

Statement	Associated keyword or symbol	Can be directly or indirectly invoked by module control, but not by test components	Can be invoked by functions, test cases and altsteps running on test components	Can be directly or indirectly invoked from specific places (see note 1)
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	
Check the state of a port	**checkstate**		Yes	
Timer operations				
Start timer	**start**	Yes	Yes	
Stop timer	**stop**	Yes	Yes	
Read elapsed time	**read**	Yes	Yes	
Check if timer running	**running**	Yes	Yes	
Timeout event	**timeout**	Yes	Yes	
Verdict operations				
Set local verdict	**setverdict**		Yes	
Get local verdict	**getverdict**		Yes	Yes
External actions				
Stimulate an (SUT) action externally	**action**	Yes	Yes	
Execution of test cases				
Execute test case	**execute**	Yes	Yes (see note 3)	

NOTE 1: Specific places are defined in clause 16.1.4. Only operations that do not have any potential side effects on snapshot evaluation are allowed.
NOTE 2: Can be used to control timer operations only.
NOTE 3: Can only be used in functions and altsteps that are used in module control.
NOTE 4: Changing of component variables is disallowed.
NOTE 5: Can be used in functions and altsteps but not in test cases.

19 Basic program statements

19.0 General

Table 16 provides an overview of the TTCN-3 basic program statements.

Table 16: 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**

Restrictions

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

a) Unless specified differently in the relevant clause, all values and templates used in a basic program statement shall be completely initialized (for exemption see e.g. clause 19.1).

NOTE: Note that the restriction applies to component of statements defined in the present document, like the boolean condition of **`if`** statements, but not to the content of statement blocks embedded into the statements.

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 that is at least partially initialized.. The effect of an assignment is to bind the variable to the value of the expression or to a template. 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. Unless the assignment is to a lazy or fuzzy variable or parameter, 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.

A structured value on the right-hand side of the assignment shall be assigned completely to the variable on the left-hand side of the assignment, If a partially initialized value is assigned to a completely initialized variable, fields uninitialized at the right-hand side of the assignment shall also become uninitialized at the left-hand side.

When a direct or indirect element or field of a lazy or fuzzy variable is assigned, the variable is also evaluated as much as necessary before assignment, i.e. if an ancestor of that element or field is initialized with a function call, it shall be evaluated. Thus, if the variable is fully assigned, it does not need to be evaluated before assignment.

NOTE: If a sub-field or sub-element of a fuzzy variable is assigned that has an ancestor which was formerly assigned a function call, this function call will be evaluated once before the assignment and replaced by its result inside the variable. Thus, the other sub-fields and sub-elements of that ancestor, apart from the field or element being assigned become non-fuzzy.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5 and shown in table 15, 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, template variable or a matching mechanism), the variable at the left hand side shall be a template variable.

c) The right-hand side of an assignment shall evaluate to an object that is at least partially initialized.

d) If the left-hand side of the assignment is a reference to a non-optional value object (i.e. a value definition, a mandatory field, a record/set of/array element, a union alternative, a value parameter), the right-hand side shall not be a reference to an omitted field or the omit symbol.

e) Using a reference to an omitted field in the right-hand side of the assignment has the same effect as using the **`omit`** keyword.

Examples

EXAMPLE 1:

```
v_myVariable := (c_x + c_y – f_increment(c_z))*3;
```

EXAMPLE 2:

```
type record MyRecord {
    record { float x, float y } c,
    integer a
}
var @lazy MyRecord v_r := {
    c := f_computeC(),
    a := f_computeA()
} // not evaluated here
v_r.c.x := f_computeX();    // first replaces field c with result of f_computeC(),
                            // then replaces field c.x with unevaluated f_computeX()
                            // field while c.y remains fixed; field a remains unevaluated
```

EXAMPLE 3:

```
type record MyRecord {
    charstring field1,
    charstring field2,
    charstring field3
}

var MyRecord v_myList1;
var MyRecord v_myList2;

v_myList1 := {"value1", "value2", "value3" };   // v_myList1 is completely initialized

v_myList2.field2 := "newvalue";                 // v_myList2 is partilly initialized
                                                // field1 and field3 remain uninitialized

v_myList1 := v_myList2;                         // v_myList1 become partially initialized,
                                                // field2 has the value "newvalue"
                                                // field1 and field3 are uninitialized
```

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 and shown in table 15.

Examples

```
if (v_date == "1.1.2005") { return ( fail ); }

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

19.3 The Select statements

19.3.1 The Select case statement

The **select case** statement is an alternative syntactic form of the **if-else** statement.

Syntactical Structure

```
select "(" SingleExpression ")" "{"
     { case "(" { TemplateInstance["," ] } ")" 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 one 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 **goto** 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.

NOTE 1: In general, it cannot be decided if templateInstances overlap or not. However, it is advised to use in the branches templateInstances that don't overlap. In such situations tools might provide better runtime performance. The handling however is tool-specific.

NOTE 2: When more than one branch could be selected (the templateInstances overlap) the textually first will be selected. For this reason overlapping is discouraged, handling however is tool-specific.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5 and shown in table 15, the following restrictions apply:

a) The **`select`** *SingleExpression* and the **`case`** *TemplateInstance*-s shall be type compatible.

b) When all templateInstances of all branches can be statically evaluated in compile time to specific values or value ranges no two branches shall match the same value.

Examples

```
select (PX_MyModulePar)  // where PX_MyModulePar is of charstring type
{
    case (charstring:"firstValue")
        {
         log ("The first branch is selected");
        }
    case (v_myCharVar, c_myCharConst)
        {
         log ("The second branch is selected");
        }
    case else
        {
         log ("The value of the module parameter PX_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 v_myLocalVar  := PX_MyModulePar;
    if (match(v_myLocalVar , charstring:"firstValue")
        {
         log ("The first branch is selected");
        }
    else if (match(v_myLocalVar , v_myCharVar) or match(v_myLocalVar , c_myCharConst))
        {
         log ("The second branch is selected");
        }
    else
        {
         log ("The value of the module parameter PX_MyModulePar is selected");
        }
}
```

19.3.2 The Select union statement

To allow easier usage of the select statement for values of union types or anytype, a special form of the select statement exists.

Syntactical Structure

```
select union "(" SingleExpression ")" "{"
     { case "(" ({ Identifier ["," ] } | { TypeIdentifier ["," ] }) ")" StatementBlock }+
     [ case else StatementBlock ]
"}"
```

Semantic Description

The statement contains a header part and one or more branches. Never more than one of the branches is executed.

In the header part of the **`select union`** statement a template instance of **`union`** type or **`anytype`** shall be given. If the template instance has a union type, each branch shall start with the **`case`** keyword followed by one or more identifiers of the alternatives (fields) of the union type (a list branch) or the **`else`** keyword (an else branch) and a statement block. If the template instance has type **`anytype`**, each branch shall start with the case keyword followed by one or more type names (a list branch) or the **`else`** keyword (an else branch) and a statement block. The StatementBlock of the list branch containing the identifier or type identifier of the chosen alternative is executed. If no case exists for the chosen alternative, the StatementBlock of the else branch, if it is present, is executed. Otherwise, the **`select union`** statement has no effect.

Restrictions

a) The *SingleExpression* in the header of the **`select union`** statement shall be of a **`union`** type. It shall be at least partially initialized.

b) Every *Identifier* in a **`case`** of the **`select union`** statement shall be an identifier of an alternative of the **`union`** type of the template instance given to the statement's header.

c) No two cases in a **`select union`** statement shall have the same case *Identifier* or *TypeIdentifier*.

Examples

```
type union Messages {
    MyMessageType1 msg1,
    MyMessageType2 msg2,
    MyMessageType3 msg3,
    MyMessageType4 msg4,
    MyMessageType5 msg5
}

function f_f(in Messages p_msg) {
    select union (p_msg) {
      case (msg1) { log(p_msg.msg1); }
      case (msg2) { log(p_msg.msg2); }
      case (msg3, msg4) { log("either msg3 or msg4"); }
      case else { log("unhandled variant"); }
}

function f_g(in anytype p_msg) {
    select union (p_msg) {
      case (integer) { log(p_msg.integer); }
      case (Messages) { f_f(p_msg.Messages); }
      case else { log("unhandled anytype variant"); }
    }
}
```

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 and shown in table 15.

Examples

```
var integer v_j;                              // Declaration of integer variable v_j
for (v_j:=1; v_j<=10; v_j:= v_j+1) { … }      // Usage of variable v_j as index variable of the
                                              // for loop


for (var float v_i:=1.0; v_i<7.9; v_i:= v_i*1.35) { … } // Index variable v_i is declared and
                                                        // initialized in the for loop header. Variable
                                                        // v_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 and shown in table 15.

Examples

```
while (v_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 and shown in table 15.

Examples

```
do { … } while (v_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 and shown in table 15, 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(mw_mySig1)
    {   label L2;                       // Definition of label L2
        pCO1.send(m_mySig2);
        pCO1.receive(mw_ySig3)
    }
[]  pCO2.receive(mw_mySig4)
    {   pCO2.send(m_mySig5);
        pCO2.send(m_mySig6);
        label L3;                       // Definition of label L3
        pCO2.receive(mw_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 and shown in table 15, 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,
m_myVar := 2 * m_myVar;
if (m_myVar < 2000) { goto L1; }         // … a jump backward to L1,
m_myVar2 := f_myFunction(m_myVar);
if (m_myVar2 > m_myVar) { goto L2; }     // … a jump forward to L2,
pCO1.send(m_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(m_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(mw_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 executed. When executed in the control part of a module or in a function called by the control part of a module, it terminates the execution of the module control part. When invoked 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 and shown in table 15.

Examples

```
module MyModule {
     :  // Module definitions
    testcase TC_MyTestCase() runs on MyMTCType system MySystemType{
        var MyPTCType v_ptc:= MyPTCType.create; // PTC creation
        v_ptc.start(f_myFunction());                // start PTC execution
         :          // test case behaviour continued
        stop        // stops the MTC, all PTCs and the whole test case
    }
    function f_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 | TemplateInstance ]
```

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 or template.

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.

The return value or template is first evaluated before returning.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5 and shown in table 15, the following restrictions apply:

a) The return statement shall not be used in testcase definitions.

Examples

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

function f_myTemplateFunction() return template charstring {
    :
    if (v_date == "1.1.2005") {
        return "2005";  // the string of the year is returned
```

```
    }
    :
    return ?;           // the any template is returned
}

function f_myBehaviour() return verdicttype {
    :
    if (f_myFunction()) {
        setverdict(pass); // use of f_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 17.

Table 17: 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 otherwise "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 17 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 and shown in table 15.

Examples

```
var integer v_myVar:= 1;
log("Line 248 in PTC_A: ", v_myVar, " (actual value of v_myVar)");
// The string "Line 248 in PTC_A: 1 (actual value of v_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 an 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 and shown in table 15.

Examples

```
do {
    …
    if (v_cond1) {
        break;          // the do-while loop is left
    }
    …
    for (var integer v_j:=1; v_j<=10; v_j:= v_j+1) {
        …
        if (v_cond2) {
            break;          // the for-loop is left but the do-while loop is continued
        }
        …
    }
    …
}
while (v_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 an error.

Restrictions

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5 and shown in table 15.

Examples

```
do {
    …
    if (v_cond) {
        continue;       // execution continues with the next iteration of the do-while-loop
    }
    …
    …
    for (var integer v_j:=1; v_j<=10; v_j:= v_j+1) {
        …
        if (v_cond2) {
            continue;   // continues with the next iteration of the for-loop
        }
```

```
        …
    }
    …
}
while (v_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 and shown in table 15.

Examples

```
var integer v_aVar:= 0;          // v_aVar is declared

{                                // start of a statement block
    var integer v_myVar:= 2;     // v_myVar is declared
    v_aVar := 5 + v_myVar;       // v_myVar is used in an assignment
}                                // end of statement block
// after leaving the statement block, v_aVar is still known, but v_myVar is not known anymore.
```

20 Statement and operations for alternative behaviours

20.0 General

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 18 summarizes the statements and operations for alternative behaviours.

Table 18: 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.

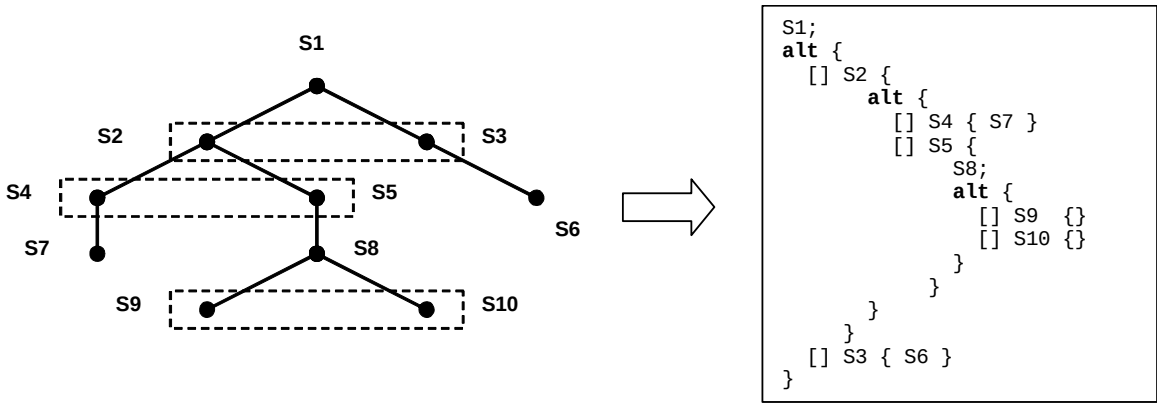


Figure 9: Illustration of alternative behaviour

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 - ETSI 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 (ETSI 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 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. An altstep-branche 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.

NOTE 2: Due to the possibility of defining dynamic test configurations, a receiving branch may refer to a disconnected or unmapped port at the time of its evaluation. In TTCN-3, ports belong to the receiving component and matching is related to the top elements in the port queues. Dynamically unmapped and disconnected ports contribute to a snapshot in the same manner as mapped and connected ports. This means, the execution of receiving operations may empty the queues of unmapped and disconnected ports without causing a test case error.

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 3: 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 4: It is also possible to use **else** in altsteps.

NOTE 5: It is allowed to use a **repeat** statement within an **else** branch.

NOTE 6: 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 and shown in table 15, 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 shall not 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 (clause 16.2) and the restrictions imposed on the contents of functions called from special places (clause 16.1.4) shall apply.

c) The evaluation of the event of an alt branch shall not have side effects. To avoid side effects that cause an inconsistency between the actual snapshot and the state of the component or introduce indeterminism in the evaluation of the following alt branches or the re-evalution of the same alt branch, the restrictions imposed on the contents of functions called from special places (clause 16.1.4) shall apply to expressions occurring in the matching part of an alternative.

d) The evaluation of an altstep invoked from an alt branch, if none of the alternatives in the altstep is chosen, shall not have side effects. To avoid side effects the restrictions imposed on the contents of functions called from special places (clause 16.1.4) shall apply to the actual parameters of the invoked altstep.

e) Void.

f) An **`alt`** statement used within the module control part shall only contain **`timeout`** statements.

Examples

EXAMPLE 1: Nested alternatives

```
alt {
    [] myPort.receive (mw_myMessage) {
        setverdict (pass);
        t_myTimer.start;
        alt {
            [] myPort.receive (mw_mySecondMessage) {
                t_myTimer.stop;
                setverdict (pass);
            }
            [] t_myTimer.timeout {
                myPort.send (m_myRepeat);
                t_myTimer.start;
                alt {
                    [] myPort.receive (mw_mySecondMessage) {
                        t_myTimer.stop;
                        setverdict (pass)
                    }
                    [] t_myTimer.timeout { setverdict (inconc) }
                    [] myPort.receive { setverdict (fail) }
                }
            }
            [] myPort.receive { setverdict (fail) }
        }
    }
    [] t_myTimer.timeout { setverdict (inconc) }
    [] myPort.receive { setverdict (fail) }
}
```

EXAMPLE 2: Alt statement with guards

```
alt {
  [v_x>1] l2.receive {                 // Boolean guard/expression
          setverdict(pass);
     }
  [v_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
        f_myErrorHandling();
        setverdict(fail);
        stop;
     }
}
```

EXAMPLE 4: Re-evaluation with repeat

```
alt {
  [] pCO3.receive {
        v_count := v_count + 1;
        repeat                  // usage of repeat
     }
  [] t_t1.timeout { }
  [] any port.receive {
        setverdict(fail);
        stop;
     }
}
```

EXAMPLE 5: Alt statement with explicitly invoked altstep

```
alt {
  [] pCO3.receive { }
  [] a_anotherAltStep() {   // Explicit call of altstep a_anotherAltStep as alternative.
        setverdict(inconc)  // Statement block executed if an alternative within
                            // altstep AnotherAltStep has been selected and executed.
     }
  [] t_myTimer.timeout { }
}
```

EXAMPLE 6: Alt statement with forbidden function calls

```
alt {
  [] f_getPort().receive(t(p())) { }  // forbidden if f_getPort, t or p has side effects
  [] a_anotherAltStep(f());           // forbidden if f has side effects
  [] t_myTimer[i(p())].timeout { }    // forbidden if i or p has side effects
  [f_g()] f_getComponent(p()).done {} // forbidden if f_g, f_getComponent or p has side effects
}
```

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 and shown in table 15, 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 {
        v_count := v_count + 1;
        repeat                  // usage of repeat
     }
  [] t_t1.timeout { }
  [] any port.receive {
        setverdict(fail);
        stop;
     }
}
```

EXAMPLE 2: Usage of repeat in an altstep

```
altstep a_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 (ETSI 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 (ETSI ES 201 873-4 [1]).

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5 and shown in table 15, the following restrictions apply:

a) Control transfer statements **activate**, **deactivate**, **repeat**, all calls of altsteps and (direct and indirect) calls of user-defined functions, which include reception statements, 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.

c) An **interleave** statement used within the module control part shall only contain **timeout** statements.

d) The restricted use of the control transfer statements **for**, **while**, **do-while**, and **goto** within **interleave** statements is allowed under the following conditions:

   a. The loop statements **for**, **while**, and **do-while** can be used within statements blocks that do not contain reception statements.

   b. The **goto** statement can be used for defining unconditional jumps within statements blocks that do not contain reception statements and for specifying unconditional jumps out of **interleave** statements.

EXAMPLE:

```
// The following TTCN-3 code fragment
interleave {
[]  pCO1.receive(mw_mySig1) {
        PCO1.send(m_mySig2);
        PCO1.receive(mw_mySig3);
    }
[]  pCO2.receive(mw_mySig4) {
        pCO2.send(m_mySig5);
        pCO2.send(m_mySig6);
        pCO2.receive(mw_mySig7);
    }
}

// is a shorthand for
alt {
[]  PCO1.receive(mw_mySig1) {
        PCO1.send(m_mySig2);
        alt {
```

```
        []  PCO1.receive(mw_mySig3) {
                alt {
                []  PCO2.receive(mw_mySig4) {
                        PCO2.send(m_mySig5);
                        PCO2.send(m_mySig6);
                        PCO2.receive(mw_mySig7)
                    }
                }
            }
        []  PCO2.receive(mw_mySig4) {
                PCO2.send(m_mySig5);
                PCO2.send(m_mySig6);
                alt {
                []  PCO1.receive(mw_mySig3) {
                        PCO2.receive(mw_mySig7);
                    }
                []  PCO2.receive(mw_mySig7) {
                        PCO1.receive(mw_mySig3);
                    }
                }
            }
        }
    }
[]  pCO2.receive(mw_mySig4) {
        pCO2.send(m_mySig5);
        pCO2.send(m_mySig6);
        alt {
        []  pCO1.receive(mw_mySig1) {
                pCO1.send(m_mySig2);
                alt {
                []  pCO1.receive(mw_mySig3) {
                        pCO2.receive(mw_mySig7);
                    }
                []  pCO2.receive(mw_mySig7) {
                        pCO1.receive(mw_mySig3);
                    }
                }
            }
        []  pCO2.receive(mw_mySig7) {
                alt {
                []  pCO1.receive(mw_mySig1) {
                        pCO1.send(m_mySig2);
                        pCO1.receive(mw_mySig3);
                    }
                }
            }
        }
    }
}
```

20.5 Default Handling

20.5.0 General

TTCN-3 allows the activation of altsteps (see clause 16.2) as defaults. For each test component the defaults, i.e. activated altsteps, are stored 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. Furthermore, the restrictions imposed on interleave statements in clause 20.4 do not apply to altsteps that are activated as default behaviour for interleave statements.

NOTE 2: Due to the possibility of defining dynamic test configurations, an alternative in an altstep activated as default may refer to a disconnected or unmapped port at the time of its evaluation. In TTCN-3, ports belong to the receiving component and matching is related to the top elements in the port queues. Dynamically unmapped and disconnected ports contribute to a snapshot in the same manner as mapped and connected ports. This means, an **altstep** invoked as default may execute receiving operations that empty the queues of unmapped and disconnected ports without causing a test case error.

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 3: 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 and shown in table 15, the following restrictions apply:

a) For altsteps activated on test components, all timer instances in the actual parameter list shall be declared as component type local timers (see clause 6.2.10.1).

b) For altsteps activated in module control or in functions or altsteps invoked directly or indirectly from module control, all timer instances in the actual parameter list shall be declared in the highest scope of the module control part (see clause 26.2). Timers from lower scopes of the module control part (i.e. from the nested statement blocks) are not allowed to occur in the actual parameter list.

c) 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 v_myDefaultVar := null;
 :
// Declaration of a default reference variable and activation of an altstep as default
var default v_myDefVarTwo := activate(a_mySecondAltStep());
 :
// Activation of altstep MyAltStep as a default
v_myDefaultVar := activate(a_myAltStep()); // a_myAltStep is activated as default
 :
// Usage of v_myDefaultVar for the deactivation of default a_myDefAltStep
deactivate(v_myDefaultVar);
```

EXAMPLE 2: Simple activation

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

EXAMPLE 3: Activation of a parameterized altstep

```
altstep a_myAltStep2 ( integer    p_value1, MyType p_value2,
                       MyPortType p_port,   timer  p_timer )
{
 :
}
function f_myFunc () runs on MyCompType
{ :
var default v_myDefaultVar := null;

v_myDefaultVar := activate(a_myAltStep2(5, v_myVar, vc_myCompPort, tc_myCompTimer);
    // MyAltStep2 is activated as default with the actual parameters 5 and
    // the value of v_myVar. A change of v_myVar before a call of a_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 and shown in table 15, 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* shall be of default type.

Examples

```
var default v_myDefaultVar := null;
var default v_myDefVarTwo := activate(a_mySecondAltStep());
var default v_myDefVarThree := activate(a_myThirdAltStep());
 :
v_myDefaultVar := activate(a_myAltStep());
 :
deactivate(v_myDefaultVar); // deactivates a_myAltStep
 :
deactivate; // deactivates all other defaults, i.e. in this case a_mySecondAltStep
            // and a_myThirdAltStep
```

21 Configuration Operations

21.0 General

Configuration operations are used to set up and control test components and their connections. They are summarized in table 19.

Restrictions

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

a) These operations shall only be used in:

- TTCN-3 test cases;
- behaviours invoked directly or indirectly from a test case or from a behaviour started on a ptc.

b) They shall not be used in:

- the module control part;
- functions or altsteps invoked directly or indirectly from the module control part;
- declarations inside component type definitions; or
- functions invoked directly or indirectly from declarations inside component type definitions.

Table 19: 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

21.1.0 General

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.

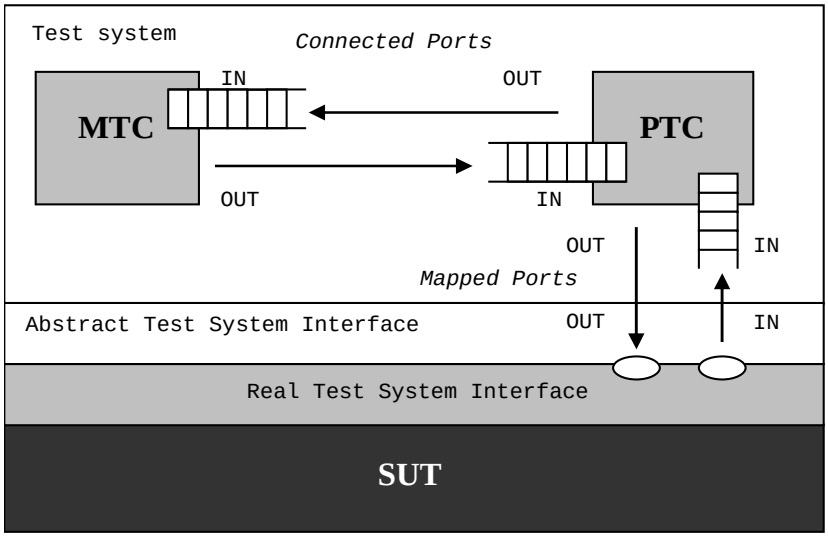


Figure 10: Illustration of the connect and map operations

21.1.1 The Connect and Map operations

The **connect** operation is used to setup connections between test components. The **map** operation are used to setup connections to the SUT.

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.

Applying a **map** or **connect** operation to ports which are already mapped or connected has no effect on the test behaviour or test configuration, i.e. test execution continues as if the operation has not been invoked.

NOTE: Please note that also triMap or tciConnect respectively will not be invoked in such a case.

The **map** operation provides an optional parameter list for configuration purposes. This allows to pass values needed for dynamic runtime configuration. If a parameter list is present, the actual parameters shall conform to the **map param** clause of the port type declaration of the system port used.

Restrictions

In addition to the general static rules of TTCN-3 given in clauses 5 and 21 and shown in table 15, 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 runtime 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.

g) In **map** operations, **param** clauses are optional. If in a **map** operation a **param** clause is present, exactly one of the components referenced by the operation shall be the **system** component reference, the type of the system component shall be known in the context of the operation either via a **system** clause or via a **runs on** clause in a **testcase** without **system** clause, the type of the system port to which the operation is applied shall include a **map param** declaration, and the actual parameters shall conform to the **map param** clause of the port type declaration of the system port used.

h) If the type of the component referenced in a connection operation is known (either when the component reference is a variable or value returned from a function or the type is defined in the runs on, mtc or system clause of the calling function), the referenced port declaration shall be present in this component type.

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  v_myNewPTC;
v_myNewPTC := MyComponentType.create;
 :
connect(v_myNewPTC:port1, mtc:port3);
map(v_myNewPTC:port2, system:pCO1);
 :
// In this example a new component of type MyComponentType is created and its reference stored
// in variable v_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 v_myConfig := { option := 1, lock := false};
 :
map(mtc:port4, system:pCO2) param (v_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.
```

EXAMPLE 3: Port visibility

```
type port P message { inout integer; }
type component C1 { port P p1; }
type component C2 { port P p1, p2; }

testcase TC runs on C1 system C1
{
  var C1 v_ptc := C2.create; // valid assignment, instance of C2 is compatible with C1 type
  connect (self:p1, v_ptc:p1); // valid, p1 is present in C1 type definition
  disconnect (self:p1, v_ptc:p1);
  connect (self:p1, v_ptc:p2); // invalid, although the real instance in v_ptc is of the
  // C2 type, the variable itself is of the C1 type making the p2 port invisible to the
  // connection operation
}
```

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 ")"
                        [ param "(" [ { ActualPar ["," ] }+ ] ")" ] )  |
        ( "(" PortRef ")" [ param "(" [ { ActualPar ["," ] }+ ] ")" ] ) |
        ( "(" ComponentRef ":" all port ")" ) |
        ( "(" all component ":" all port ")" ) ]
```

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. If a parameter list is present, the actual parameters shall 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

In addition to the general static rules of TTCN-3 given in clauses 5 and 21 and shown in table 15, the following restrictions apply:

a) In an **unmap** operation, a **param** clause shall only be present if the system port to which the **param** clause belongs to is explicitly referenced.

b) In **unmap** operations, **param** clauses are optional. If in an **unmap** operation a **param** clause is present, exactly one of the components referenced by the operation shall be the **system** component reference, the type of the system component shall be known in the context of the operation either via a **system** clause or via a **runs on** clause in a **testcase** without **system** clause, the type of the system port to which the operation is applied shall include an **unmap param** declaration and the actual parameters shall conform to the **unmap param** clause of the port type declaration of the system port used.

c) If the type of the component referenced in a connection operation is known (either when the component reference is a variable or value returned from a function or the type is defined the runs on, mtc or system clause of the calling function), the referenced port declaration shall be present in this component type.

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

21.2.0 General

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

No specific restrictions in addition to the general static rules of TTCN-3 given in clause 5 and shown in table 15.

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

21.3.0 General

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 only once and a PTC that is kept alive after termination of a behaviour and can be therefore reused to execute another behaviour. The latter is created using the additional **alive** keyword. An alive-type PTC shall be destroyed explicitly using the **kill** operation (see clause 21.3.4), whereas a non-alive PTC is destroyed implicitly after its behaviour 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 clauses 5 and 21 and shown in table 15, 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 v_myNewComponent;
var MyComponentType v_myNewestComponent;
var MyComponentType v_myAliveComponent;
var MyComponentType v_myAnotherAliveComponent;
var MyComponentType v_myDeployedComponent;
 :
v_myNewComponent := MyComponentType.create;
v_myNewestComponent := MyComponentType.create("Newest");
v_myAliveComponent := MyComponentType.create alive;
v_myAnotherAliveComponent := MyComponentType.create("Another Alive") alive;
v_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 | AltstepInstance ) ")"
```

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 or altstep.

An alive-type PTC may perform several behaviours in sequential order. Starting a second behaviour on a non-alive PTC or starting a behaviour on a PTC that is still running results in a test case error. If a behaviour is started on an alive-type PTC after termination of a previous behaviour, it uses variable values, timers, ports, and the local verdict as they were left after termination of the previous behaviour. In particular, if a timer was started in the previous behaviour, the subsequent behaviour 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 behaviour with a corresponding **runs on**-clause that is started on an alive-type PTC.

Actual inout parameters will be passed to the function by value, i.e. like in-parameters.

If the function's formal parameter list includes any out parameter the actual parameter list may omit actual out parameters using the dash symbol ("-") or be omitted in the same manner as for actual in parameters with default values (see clause 5.4.2), i.e. they can be omitted in the list notation if all following actual parameters are also omitted and their assignment can be omitted altogether in assignment notation. If a variable is given as an actual out parameter, it will remain unchanged by the started behaviour, even if the behaviour changes the formal parameter during its execution.

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.

Restrictions

In addition to the general static rules of TTCN-3 given in clauses 5 and 21 and shown in table 15, 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 the first *FunctionInstance* shall be of component type.

b) The function or altstep invoked in a **start** test component operation shall have a **runs on** definition referencing a component type that is compatible with the newly created component (see clause 6.3.3).

c) Ports, defaults and timers shall not be passed into a function or altstep invoked in a **start** test component operation. All formal parameter types of the behaviour shall neither be of port or default type or should contain a direct or indirect element or field of port or default type.

NOTE 2: 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 f_myFirstBehaviour() runs on MyComponentType { … }
function f_mySecondBehaviour() runs on MyComponentType { … }
function f_myThirdBehaviour(out integer p_p1, inout integer p_p2) runs on MyComponentType { … }
altstep  a_myFourthBehaviour() runs on MyComponentType { ... }
:
var MyComponentType v_myNewPTC;
var MyComponentType v_myAlivePTC;
var integer v_int := 0;
:
v_myNewPTC := MyComponentType.create;           // Creation of a new non-alive test component.
v_myAlivePTC := MyComponentType.create alive;   // Creation of a new alive-type test component
:
v_myNewPTC.start(f_myFirstBehaviour());     // Start of the non-alive component.
v_myNewPTC.done;                            // Wait for termination
v_myNewPTC.start(f_mySecondBehaviour());    // Test case error
:
v_myAlivePTC.start(f_myFirstBehaviour());   // Start of the alive-type component
v_myAlivePTC.done;                          // Wait for termination
v_myAlivePTC.start(f_mySecondBehaviour());  // Start of the next function on the same component
:
v_myAlivePTC.start(f_myThirdBehaviour(-,v_int));    // v_int will not be changed by the function
v_myAlivePTC.done;
v_myAlivePTC.start(a_myFourthBehaviour());      // Direct start of an altstep behaviour<>
```

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 test behaviour 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 clauses 5 and 21 and shown in table 15, 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* shall be of component type.

Examples

EXAMPLE 1: Stopping another test component and a test component by itself

```
var MyComponentType v_myComp := MyComponentType.create; // A new test component is created
v_myComp.start(f_compBehaviour());                       // The new component is started
:
if (v_date == "1.1.2005") {
    v_myComp.stop;                 // The component "v_myComp" is stopped
}

:
if (v_a < v_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 and 21 and shown in table 15, 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* shall be of component type.

Examples

EXAMPLE 1: Killing another test component and a test component by itself

```
var PTCType v_myAliveComp := PTCType.create alive;  // Create an alive-type test component
v_myAliveComp.start(f_myFirstBehaviour());          // The new component is started
v_myAliveComp.done;                                 // Wait for termination
v_myAliveComp.start(f_mySecondBehavior());          // Start the component a 2nd time
v_myAliveComp.done;                                 // Wait for termination
v_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.

Syntactical Structure

```
( VariableRef |
  FunctionInstance |
  any component |
  all component |
  any from ComponentArrayRef ) "." alive
[ "->" @index value VariableRef ]
```

Semantic Description

Applied on a normal parallel test component, the **alive** operation returns true if the component is inactive or running a behaviour and false otherwise. Applied on an alive-type parallel test component, the operation returns true if the component is inactive, running or stopped. It returns false if the component has been killed. Applied on the **mtc** the operation returns **true**.

The **alive** operation can be used similar to the **running** operation (see clause 21.3.6). 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.

When the **any from** component array notation is used, the components from the referenced array are iterated over and individually checked for being inactive or running a function from innermost to outermost dimension from lowest to highest index for each dimension. The first component to be found being inactive or running a behaviour causes the alive operation to return the **true** value. The index of the first component found alive can optionally be assigned to an integer variable for single-dimensional component arrays or to an integer array or record of integer variable for multi-dimensional component arrays.

Restrictions

In addition to the general static rules of TTCN-3 given in clauses 5 and 21 and shown in table 15, 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* shall be of component type.

b) The *ComponentArrayRef* shall be a reference to a completely initialized component array.

c) The index redirection shall only be used when the operation is used on an **any from** component array construct.

d) If the index redirection is used for single-dimensional component arrays, the type of the integer variable shall allow storing the highest index of the respective array.

e) If the index redirection is used for multi-dimensional component arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective array, and its type shall allow storing the highest index (from all dimensions) of the array.

f) If a variable referenced in the **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **alive** operation, i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **alive** operation.

Examples

```
pTC1.done;                          // Waits for termination of the component
if (pTC1.alive) {                   // If the component is still alive …
    pTC1.start(f_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.

Syntactical Structure

```
( VariableRef |
  FunctionInstance |
  any component |
  all component |
  any from ComponentArrayRef ) "." running
[ "->" @index value VariableRef ]
```

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 returns **true** for the **mtc** and 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**.

NOTE: The difference between the **running** operation applied to a single ptc and the usage of the **all** keyword leads to the situation that **ptc.running** is **false** if the ptc has never been started but **all component.running** is **true** at the same time as it considers only those components that ever have been started.

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**.

When the **any from** component array notation is used, the components from the referenced array are iterated over and individually checked for executing currently from innermost to outermost dimension from lowest to highest index for each dimension. The first component to be found executing causes the running operation to succeed. The index of the matched component can optionally be assigned to an integer variable for single-dimensional arrays or to an integer array or record of integer variable for multi-dimensional component arrays.

Restrictions

In addition to the general static rules of TTCN-3 given in clauses 5 and 21 and shown in table 15, 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* shall be of component type.

b) The *ComponentArrayRef* shall be a reference to a completely initialized component array.

c) The index redirection shall only be used when the operation is used on an **any from** component array construct.

d) If the index redirection is used for single-dimensional component arrays, the type of the integer variable shall allow storing the highest index of the respective array.

e) If the index redirection is used for multi-dimensional component arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective array, and its type shall allow storing the highest index (from all dimensions) of the array.

f) If a variable referenced in the **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **running** operation. Later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **running** operation.

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
    f_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. In addition, the **done** operation allows to retrieve the final local verdict of completed test components, i.e., the value of the local verdict at the time of test component completion.

Syntactical Structure

```
( VariableRef |
  FunctionInstance |
  any component |
  all component |
  any from ComponentArrayRef ) "." done
[ "->" [ value VariableRef ] [ @index value VariableRef ] ]
```

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.

NOTE 1: The execution of a **done** operation does not change the state of the test component. Consecutive **done** operations applied to the same test component will give the same result as long as the test component does not change its state (see clause F.1.2).

When the **done** operation is applied to a PTC and matches, the final local verdict of the PTC can be retrieved and stored in variable of the type **verdicttype**. This is denoted by the symbol '**->**' the keyword **value** followed by the name of the variable into which the verdict is stored.

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.

NOTE 2: The difference between the **done** operation applied to a single ptc and the usage of the **all** keyword leads to the situation that **ptc.done** does not match if the ptc has never been started but **all component.done** matches at the same time as it considers only those components that ever have been started.

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 3: 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.

When the **any from** component array notation is used, the components from the referenced array are iterated over and individually checked for being stopped or killed from innermost to outermost dimension from lowest to highest index for each dimension. The first component to be found stopped or killed causes done operation to succeed. The index of the matched component can optionally be assigned to an integer variable for single-dimensional arrays or to an integer array or record of integer variable for multi-dimensional component arrays.

Restrictions

In addition to the general static rules of TTCN-3 given in clauses 5 and 21 and shown in table 15, the following restrictions apply:

a) The **done** operation can be used for PTCs only.

b) The variable or the return type associated with *FunctionInstance* followed by the **done** keyword, i.e. used for identifying a specific PTC, shall be of component type.

c) The *ComponentArrayRef* shall be a reference to a completely initialized component array.

d) The variable used in the (optional) **value** clause for storing the final local verdict of a PTC shall be of the type **verdicttype**.

e) The (optional) **value** clause for storing the final local verdict of a PTC shall not be used in combination with **all component** or **any component**.

f) The index redirection shall only be used when the operation is used on an **any from** component array construct.

g) If the index redirection is used for single-dimensional component arrays, the type of the integer variable shall allow storing the highest index of the respective array.

h) If the index redirection is used for multi-dimensional component arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective array, and its type shall allow storing the highest index (from all dimensions) of the array.

i) If a variable referenced in the **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **done** operation. Later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **done** operation.

Examples

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

    []  any port.receive {
            repeat
        }
}

var MyComp v_c := MyComp.create alive;
v_c.start(f_myPTCBehaviour());
:
v_c.done;
    // matches as soon as the function f_myPTCBehaviour (or function/altstep called by it) stops
v_c.done;
    // matches again, even if the component has not been started again
if(v_c.running) {v_c.done}
    // in case that some other component has started v_c in the meantime
    // done here matches the end of the next behaviour only, not the previous one

// 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

// Retrieving and using the final local verdict of a completed PTC
var MyComp v_myPTC := MyPTC.create alive;
var verdicttype v_myPTCverdict := none;
v_myPTC.start(f_myPTCBehaviour());
:
```

```
alt {
    []  v_myPTC.done -> value v_myPTCverdict {
            if (v_myPTCverdict == fail) {
                setverdict(fail);
                stop;
            }
            else {
                setverdict (pass);
            }
        }

    []  any port.receive {
            repeat
        }
}
```

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. In addition, the **killed** operation allows to retrieve the final local verdict of killed test components, i.e., the value of the local verdict at the time when the test component was killed.

Syntactical Structure

```
( VariableRef |
  FunctionInstance |
  any component |
  all component |
  any from ComponentArrayRef ) "." killed
[ "->" [ value VariableRef ] [ @index value VariableRef ] ]
```

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 1: 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 21.3.7). 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.

NOTE 2: The execution of a **killed** operation does not change the state of the test component. Consecutive **killed** operations applied to the same test component will give the same result as long as the test component does not change its state (see clause F.1.2).

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 **killed** operation is applied to a PTC and matches, the final local verdict of that PTC can be retrieved and stored in a variable of the type **verdicttype**. This is denoted by the symbol '**->**' the keyword **value** followed by the name of the variable into which the verdict is stored.

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.

When the **any from** component array notation is used, the components from the referenced array are iterated over and individually checked for being killed from innermost to outermost dimension from lowest to highest index for each dimension. The first component to be found killed causes the killed operation to succeed. The index of the matched component can optionally be assigned to an integer variable for single-dimensional component arrays or to an integer array or record of integer variable for multi-dimensional component arrays.

Restrictions

In addition to the general static rules of TTCN-3 given in clauses 5 and 21 and shown in table 15, the following restrictions apply:

a) The **killed** operation can be used for PTCs only.

b) The variable or the return type associated with *FunctionInstance* followed by the **killed** keyword, i.e. used for identifying a specific PTC, shall be of a component type.

c) The *ComponentArrayRef* shall be a reference to a completely initialized component array.

d) The variable used in the (optional) **value** clause for storing the final local verdict of a PTC shall be of the type **verdicttype**.

e) The (optional) **value** clause for storing the final local verdict of a PTC shall not be used in combination with **all component** or **any component**.

f) The index redirection shall only be used when the operation is used on an **any from** component array construct.

g) If the index redirection is used for single-dimensional component arrays, the type of the integer variable shall allow storing the highest index of the respective array.

h) If the index redirection is used for multi-dimensional component arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective array, and its type shall allow storing the highest index (from all dimensions) of the array.

i) If a variable referenced in the **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **killed** operation i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **killed** operation.

Examples

```
var MyPTCType v_ptc := MyPTCType.create alive;  // create an alive-type test component
timer t_T:= 10.0;                               // create a timer
t_T.start;                                      // start the timer
v_ptc.start(f_myTestBehavior());                // start executing a function on the PTC
alt {
[] v_ptc.killed {                               // if the PTC was killed during execution …
    t_T.stop;                                   // … stop the timer and …
    setverdict(inconc);                         // … set the verdict to 'inconclusive'
   }
[] v_ptc.done {                                 // if the PTC terminated regularly …
    t_T.stop;                                   // … stop the timer and …
    v_ptc.start(f_anotherFunction());           // … start another function on the PTC
   }
[] t_T.timeout {                                // if the timeout occurs before the PTC stopped
    v_ptc.kill;                                 // … kill the PTC and …
    setverdict(fail);                           // … set the verdict to 'fail'
   }
}

// Retrieving and using the final local verdict of a killed PTC
var MyComp v_myPTC := MyPTC.create alive;
var verdicttype v_myPTCverdict := none;
v_myPTC.start(f_myPTCBehaviour());
:
alt {
    []  v_myPTC.done {                          // expected termination
            setverdict (pass);
        }
    }
    []  v_myPTC.killed -> value v_myPTCverdict {
            if (v_MyPTCverdict == none) {       // v_myPTC killed before verdict assingment
                setverdict(fail);
                stop;
            }
            else {
                setverdict (inconc);            // further analysis is needed
                stop;
            }
```

```
        }
    [] any port.receive {
        repeat
        }
}
```

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 20.

Table 20: 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

22.0 General

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 21.

Table 21: 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
Check the state of a port	**checkstate**	Yes	Yes

22.1 The communication mechanisms

22.1.0 General

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.

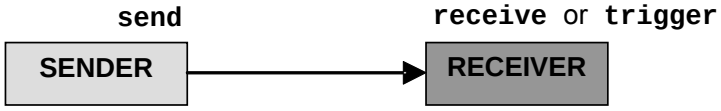


Figure 11: Illustration of the asynchronous send, receive and trigger

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.

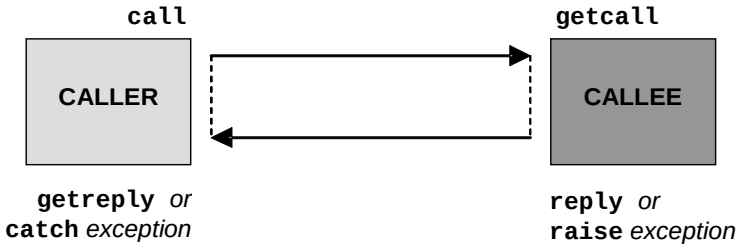


Figure 12: Illustration of blocking procedure-based communication

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.

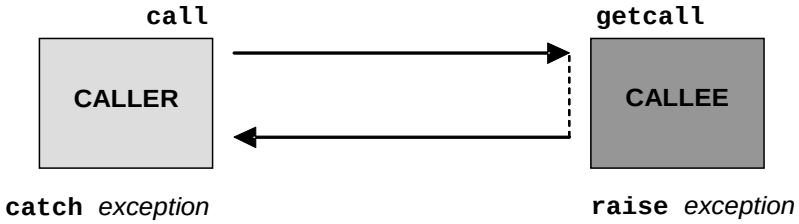


Figure 13: Illustration of non-blocking procedure-based communication

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

22.1.4.0 General

Operations such as **send** and **call** are used for the exchange of information among test components and between an SUT and test components. For explaining the general format of these operations, they can be structured into two groups:

a) a test component sends a message (**send** operation), calls a procedure (**call** operation), or replies to an accepted call (**reply** operation) or raises an exception (**raise** operation). These actions are collectively referred to as *sending operations*;

b) a component receives a message (**receive** operation), awaits a message (**trigger** operation),accepts a procedure call (**getcall** operation), receives a reply for a previously called procedure (**getreply** operation) or catches an exception (**catch** operation). These actions are collectively referred to as *receiving operations*.

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 sent.

The port name, operation name and value shall be present in all sending operations. The address part (denoted by the **to** keyword) is optional and need only be specified in cases of one-to-many connections where:

- unicast communication is used and one receiving entity shall be explicitly identified;
- multicast communication is used and a set of receiving entities has to be explicitly identified;
- broadcast communication is used and all entities connected to the specified port have to be addressed.

EXAMPLE 1:

Send part			**(Optional) response and exception handling part**
Port and operation	**Value part**	**(Optional) address part**	
`myP1.`**`send`**	`(v_myVariable + v_yourVariable - 2)`	**`to`** `v_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:{s_myVar1})`		`{`
			`  [] myP1.`**`getreply`**`(MyProc:{s_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 message or parameter value assignment part strong typing is not required, e.g. the variable used for storing a message shall be type-compatible to the type of 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 shall be sent back to the original sending component.

For receiving operations using the any port from a port array construction (see clause 22.2.2), the assignment part may also be used to store the indices that identify the specific port instance where the receiving operation matched.

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`** `(v_v1)`	**`sender`** `v_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`**	`(mw_myTemplate(5,7))`	**`from`** `v_aPeer`	`->`	**`value`** `v_myVar`		

Receive part				(Optional) assignment part			
Port and operation	**Matching part**	**(Optional) address expression**		**(Optional) value assignment**	**(Optional) parameter value assignment**	**(Optional) sender value assignment**	**(Optional) port index assignment**
`any from` `p.`**`receive`**	`(mw_myTemplate(5,7))`		`->`				**`@index value`** `v_i`

22.2 Message-based communication

22.2.0 General

The operations for message-based communication via asynchronous ports are summarized in table 22.

Table 22: 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 and shown in table 15, 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) All *AddressRef* items in the **to** clause shall be of type **address**, **component** or of the address type bound to the port type (see clause 6.2.9) of the port instance referenced in the **send** operation. No *AddressRef* in the **to** clause shall contain the special value **null** at the time of the operation.

f) Applying a **send** operation to an unmapped or disconnected port shall cause a test case error.

Examples

EXAMPLE 1: Simple send (receiver is determined from the test configuration)

```
myPort.send(m_myTemplate(5,v_myVar));   // Sends the template m_myTemplate with the actual
                                        // parameters 5 and v_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 v_myPartner;
                                        // Sends the string "My string" to a component with a
                                        // component reference stored in variable v_myPartner

myPCO.send(v_myVariable + v_yourVariable - 2) to v_myPartner;
                                // Sends the result of the arithmetic expression to v_myPartner.

myPCO2.send(m_myTemplate) to (v_myPeerOne, v_myPeerTwo);
                                // Specifies a multicast communication, where the value of
                                // m_myTemplate is sent to the two component references stored
                                // in the variables v_myPeerOne and v_myPeerTwo.

myPCO3.send(m_myTemplate) to all component;
                                // Broadcast communication: the value of m_mytemplate is sent to
                                // all components which can be addressed via this port. If
                                // myPCO3 is a mapped port, the components may reside inside
                                // the SUT.
```

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 | any from PortArrayRef ) "." receive
[ "(" TemplateInstance ")" ]
[ from Address ]
[ "->" [ value ( VariableRef |
                 ( "(" { VariableRef [ ":=" [ @decoded [ "(" Expression ")" ] ]
                         FieldOrTypeReference ][","] } ")" )
               ) ]
       [ sender VariableRef ]
       [ @index value 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 **alt** statement shall continue with its next alternative.

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 followed by a specification of an address or component reference, a list of address or component references or **any component**.

NOTE 3: The one-to-one connection is considered to be a simple case of the one-to-many connections and allows the usage of the **from**-clause.

Storing the received message and parts of the received message

If the match is successful, the value is 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 a list enframed by a pair of parentheses, the whole received message and/or one or more parts of it can be stored. For each list element that consists only of a variable or formal parameter name the whole message shall be stored in that variable or formal parameter. The type of the variable or formal parameter shall be compatible with the type of the message. Each assignment notation member of the list allows storing the value of the field or element of the received message, which is referenced on the right hand side of the assignment notation (:=), in the variable or formal parameter on the left hand side. The variable or formal parameter shall be type compatible with the type of the referenced field or element.

When assigning individual fields of a message, encoded payload fields can be decoded prior to assignment using the **@decoded** modifier. In this case, the referenced field on the right hand side of the assignment shall be one of the **bitstring**, **hexstring**, **octetstring**, **charstring** or **universal charstring** types. It shall be decoded into a value of the same type as the variable on the left hand side of the assignment. Failure of this decoding shall cause a test case error. In case the referenced field is of the **universal charstring** type, the **@decoded** clause can contain an optional parameter defining the encoding format. The parameter shall be of the **charstring** type and it shall contain one of the strings allowed for the **decvalue_unichar** function (specified in clause C.5.4). Any other value shall cause an error. In case the referenced field is not a **universal charstring**, the optional parameter shall not be present.

NOTE 4: The model of the behaviour of this implicit decoding is defined in clause B.1.2.9.

NOTE 5: The **@decoded** clause is typically used together with the **decmatch** matching mechanism in the matching part of the receive statement. Since the decoding procedures for assignment and matching are virtually the same, TTCN-3 tools can be optimized in such a way that only one call to the decoder is made when the receiving statement contains both **decmatch** matching mechanism and **@decoded** assignment for the same payload field.

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.

Receive on any port from a port array

To **receive** a message on any port from a specific port array, use the **any from** *PortArrayRef* syntax where PortArrayRef shall be a reference to a port array identifier**.** It is also possible to store the index of a port in a single-dimensional port array at which the operation was successful to a variable of type integer or, in case of multi-dimensional port arrays the index of the successful port to an integer array or record of integer variable. When checking the port array for matching messages, the port indices to be checked are iterated from lowest to highest. If the port array is multi-dimensional, then the ports are iterated over from innermost to outermost array dimension from lowest to highest index for each dimension, e.g. [0][0], [0][1], [1][0], [1][1]. The first port which matches all the criteria will cause the operation to be successful even if other ports in the array would also meet the criteria.

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 and shown in table 15, 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.

NOTE 6: An error due to a type mismatch may happen if the types in the receive part are not compatible to the types in the assignment part or, if the **from** clause is missing, but the type of the sender can be determined and it is not type compatible with the type in the **sender** clause.

f) All *AddressRef* items in the **from** clause and all *VariableRef* items in the **sender** clause shall be of type **address**, **component** or of the address type bound to the port type (see clause 6.2.9) of the port instance referenced in the **receive** operation. No *AddressRef* in the **from** clause shall contain the special value **null** at the time of the operation.

g) The *PortArrayRef* shall be a reference to a completely initialized port array.

h) The index redirection shall only be used when the operation is used on an any from port array construct.

i) If the index redirection is used for single-dimensional port arrays, the type of the integer variable shall allow storing the highest index of the respective array.

j) If the index redirection is used for multi-dimensional port arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective array, and its type shall allow storing the highest index (from all dimensions) of the array.

k) If a variable referenced in the **value**, **sender** or **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **receive** operation i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **receive** operation.

l) If the **receive** operation contains both **from** and **sender** clause, the variable or parameter referenced in the **sender** clause shall be type compatible with the template in the **from** clause.

m) When assigning implicitly decoded message fields (by using the **@decoded** modifier) in cases where the value or template to be matched uses the *MatchDecodedContent* (**decmatch**) matching for the field to be stored, the type of the template in the *MatchDecodedContent* matching shall be type-compatible to the type of the variable the decoded field is stored into.

Examples

EXAMPLE 1: Basic receive

```
myPort.receive(mw_myTemplate(5, v_myVar));  // Matches a message that fulfils the conditions
                                            // defined by template mw_myTemplate at port myPort.

myPort.receive(v_a<v_b);    // Matches a Boolean value that depends on the outcome of v_a<v_b

myPort.receive(integer:v_myVar);    // Matches an integer value with the value of v_myVar
                                    // at port myPort

myPort.receive(v_myVar);                // Is an alternative to the previous example
```

EXAMPLE 2: Receiving from a sender, storing the message, parts of the message or the sender

```
type MyPayloadType record {
  integer       messageId,
  ContentType   content
}
type MyType2 record {
  Header        header,
  octetstring   payload
}

template MyType mw_myTemplate := {
  messageId := 42,
  content := ?
}
...
var MyPayloadType v_myVar;
var integer v_myMessageIdVar, v_myIntegerVar;
var charstring v_myCharstringVar;
var address v_myPeer;
var octetstring v_myVarOne := '00ff'O;

MyPort.receive(charstring:"Hello")from v_myPeer;    // Matches charstring "Hello" from MyPeer

MyPort.receive(MyType:?) -> value v_myVar;  // The value of the received message is
                                            // assigned to v_myVar.

MyPort.receive(MyType:?) -> value (v_myVar, v_myMessageIdVar:= messageId)
                            // The value of the received message is stored in the variable
                            // v_myVar and the value of the messageId field of the received
                            // message is stored in the variable v_myMessageIdVar.

MyPort.receive(anytype:?) -> value (v_myIntegerVar:= integer)
                            // If the received value is an integer, it is stored in the variable
                            // v_myIntegerVar, a test case error otherwise.

MyPort.receive(charstring:?) -> value (v_myCharstringVar)
                            // The received value is stored in the variable v_myCharstringVar;
                            // Note that it is the same as to write "value v_myCharstringVar"

MyPort.receive(A<B) -> sender v_myPeer;     // The address of the sender is assigned to v_myPeer

MyPort.receive(MyType:{5, v_myVarOne }) -> value v_myVar sender v_myPeer;
// The received message value is stored in v_myVar and the sender address is stored in
// v_myPeer.
MyPort.receive(MyType2:{header := ?, payload := decmatch mw_myTemplate})
    -> value (v_myVar := @decoded payload);
// The encoded payload field of the received message is decoded and matched with
// mw_myTemplate; if the matching is successful the decoded payload is stored in v_myVar.
```

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 v_mySenderVar; // Removes the top message from myPort and assigns
                                         // the sender address to v_mySenderVar
```

EXAMPLE 4: Receive on any port

```
any port.receive(mw_myMessage);
```

EXAMPLE 5: Receive on any port from a port array

```
type port MyPort message { inout integer }
type component MyComponent {
  port MyPort p[10][10];
}
var integer v_i[2];
any from p.receive(mw_myMessage) -> @index value v_i;
// checking receiving mw_myMessage on any port of the port array p and storing the index of the
// port on which the matching was successful first; if, for example MyMessage is matched first
// on p[4,2], the content of i will be {4,2}
```

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 | any from PortArrayRef ) "." trigger
[ "(" TemplateInstance ")" ]
[ from Address ]
[ "->" [ value ( VariableRef |
                 ( "(" { VariableRef [ ":=" [ @decoded [ "(" Expression ")" ] ]
                         FieldOrTypeReference ][","] } ")" )
               ) ]
  [ sender VariableRef ]
  [ @index value VariableRef ] ]
```

NOTE 1: *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 from a specific sender

In the case of one-to-many connections the **trigger** operation may be restricted to a certain communication partner. This restriction shall be denoted using the **from** keyword followed by a specification of an address or component reference, a list of address or component references or **any component**.

NOTE 2: The one-to-one connection is considered to be a simple case of the one-to-many connections and allows the usage of the **from**-clause.

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.

Trigger on any port from a port array

To trigger on a message at any port from a specific port array, use the **any from** *PortArrayRef* syntax where PortArrayRef shall be a reference to a port array identifier**.** It is also possible to store the index of a port in a single-dimensional port array at which the operation was successful to a variable of type integer or, in case of multi-dimensional port arrays the index of the successful port to an integer array or record of integer variable. When checking the port array for matching messages, the port indices to be checked are iterated from lowest to highest. If the port array is multi-dimensional, then the ports are iterated over from innermost to outermost array dimension from lowest to highest index for each dimension, e.g. [0][0], [0][1], [1][0], [1][1]. The first port which matches all the criteria will cause the operation to be successful even if other ports in the array would also meet the criteria.

If any port in the port array which is checked for matching contains a message that does not match, this message is removed and the containing **alt** statement is re-evalutated, regardless of whether or not other ports in the port array would meet the trigger criteria.

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 ETSI 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 and shown in table 15, 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.

NOTE 3: An error due to a type mismatch may happen if the types in the receive part are not compatible to the types in the assignment part or, if the **from** clause is missing, but the type of the sender can be determined and it is not type compatible with the type in the **sender** clause.

d) All *AddressRef* items in the **from** clause and all *VariableRef* items in the **sender** clause shall be of type **address**, **component** or of the address type bound to the port type (see clause 6.2.9) of the port instance referenced in the **trigger** operation. No *AddressRef* in the **from** clause shall contain the special value **null** at the time of the operation.

e) The *PortArrayRef* shall be a reference to a completely initialized port array.

f) The index redirection shall only be used when the operation is used on an any from port array construct.

g) If the index redirection is used for single-dimensional port arrays, the type of the integer variable shall allow storing the highest index of the respective array.

h) If the index redirection is used for multi-dimensional port arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective array, and its type shall allow storing the highest index (from all dimensions) of the array.

i) If a variable referenced in the **value**, **sender** or **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **trigger** operation, i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **trigger** operation.

j) If the **trigger** operation contains both **from** and **sender** clause, the variable or parameter referenced in the **sender** clause shall be type compatible with the template in the **from** clause.

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 v_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 v_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 v_myVar sender v_myPartner;
// Trigger on the reception of an arbitrary integer value which afterwards is stored in
// variable v_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 v_mySenderVar;
```

EXAMPLE 4: Trigger on any port

```
any port.trigger
```

EXAMPLE 5: Trigger on any port from port array

```
type port MyPort message { inout integer }
type component MyComponent {
  port MyPort p[10][10];
}
var integer v_i[2];
any from p.trigger(mw_myMessage) -> @index value v_i;
// Checking if mw_myMessage has been received on any port of the port array p; if yes, the index
// of the port on which the matching was first successful is stored in the array v_i; if no port
// succeeds, the top messages are removed and the port array is re-checked.
```

22.3 Procedure-based communication

22.3.0 General

The operations for procedure-based communication via synchronous ports are summarized in table 23.

Table 23: 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 and shown in table 15, 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 shall 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) All *AddressRef* items in the **to** clause shall be of type **address**, **component** or of the address type bound to the port type (see clause 6.2.9) of the port instance referenced in the **call** operation. No *AddressRef* in the **to** clause shall contain the special value **null** at the time of the operation.

g) *CallTimerValue* shall 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.

m) Applying a **call** operation to an unmapped or disconnected port shall cause a test case error.

Examples

EXAMPLE 1: Blocking call with getreply

```
// Given …
signature MyProc (out integer MyPar1, inout boolean MyPar2);
 :
// a call of MyProc
myPort.call(MyProc:{ -, v_myVar2}) {        // in-line signature template for the call of MyProc
    [] myPort.getreply(MyProc:{?, ?}) { }
}

// … and another call of MyProc
myPort.call(s_myProcTemplate) {             // using signature template for the call of MyProc
    [] myPort.getreply(MyProc:{?, ?}) { }
}

myPort.call(s_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 v_myResult param (v_myPar1Var,v_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,v_myVar}, 20E-3) {

  [] myPort.getreply(MyProc:{?, ?}) { }

  [] myPort.catch(timeout) {               // timeout exception after 20ms
       setverdict(fail);
       stop;
     }
}
```

EXAMPLE 4: Nowait call

```
myPort.call(MyProc:{5, v_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 v_first:= true;
myPort.call(MyProc:{5,v_myVar}, 20E-3) to all component {   // Broadcast call of MyProc
    // Handles the response from myPeerOne
    [v_first] myPort.getreply(MyProc:{?, ?}) from myPeerOne {
        if (v_first) { v_first := false; repeat; }
        :
    }
    // Handles the response from myPeerTwo
    [v_first] myPort.getreply(MyProc:{?, ?}) from myPeerTwo {
        if (v_first) { v_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,v_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 | any from PortArrayRef ) "." getcall
[ "(" TemplateInstance ")" ]
[ from Address ]
[ "->" [ param "(" { ( VariableRef ":=" [ @decoded [ "(" Expression ")" ] ]
                     ParameterIdentifier ) "," } |
                     { ( VariableRef | "-" ) "," }
                   ")" ]
       [ sender VariableRef ]
       [ @index value VariableRef ] ]
```

NOTE 1: *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 followed by a specification of an address or component reference, a list of address or component references or **any component**.

NOTE 2: The one-to-one connection is considered to be a simple case of the one-to-many connections and allows the usage of the **from**-clause.

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.

When assigning individual parameters of a call, encoded parameters can be decoded prior to assignment using the **@decoded** modifier. In this case, the referenced parameter on the right hand sided of the assignment shall be one of the **bitstring**, **hexstring**, **octetstring**, **charstring** or **universal charstring** types. It shall be decoded into a value of the same type as the variable on the left hand side of the assignment. Failure of this decoding shall cause a test case error. In case the referenced field is of the **universal charstring** type, the **@decoded** clause can contain an optional parameter defining the encoding format. The parameter shall be of the **charstring** type and it shall contain one of the strings allowed for the **decvalue_unichar** function (specified in clause C.5.4). Any other value shall cause an error. In case the referenced field is not a **universal charstring**, the optional parameter shall not be present.

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.

Getcall on any port from a port array

To **getcall** on any port from a specific port array, use the **any from** *PortArrayRef* syntax where PortArrayRef shall be a reference to a port array identifier**.** It is also possible to store the index of a port in a single-dimensional port array at which the operation was successful to a variable of type integer or, in case of multi-dimensional port arrays the index of the successful port to an integer array or record of integer variable. When checking the port array for matching calls, the port indices to be checked are iterated from lowest to highest. If the port array is multi-dimensional, then the ports are iterated over from innermost to outermost array dimension from lowest to highest index for each dimension, e.g. [0][0], [0][1], [1][0], [1][1]. The first port which matches all the criteria will cause the operation to be successful even if other ports in the array would also meet the criteria.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5 and shown in table 15, 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 shall 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) All *AddressRef* items in the **from** clause and all *VariableRef* items in the **sender** clause shall be of type **address**, **component** or of the address type bound to the port type (see clause 6.2.9) of the port instance referenced in the **getcall** operation. No *AddressRef* in the **from** clause shall contain the special value **null** at the time of the operation.

g) The *PortArrayRef* shall be a reference to a completely initialized port array.

h) The index redirection shall only be used when the operation is used on an any from port array construct.

i) If the index redirection is used for single-dimensional port arrays, the type of the integer variable shall allow storing the highest index of the respective array.

j) If the index redirection is used for multi-dimensional port arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective array, and its type shall allow storing the highest index (from all dimensions) of the array.

k) If a variable referenced in the **param**, **sender** or **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **getcall** operation, i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **getcall** operation.

l) If the **getcall** operation contains both **from** and **sender** clause, the variable or parameter referenced in the **sender** clause shall be type compatible with the template in the **from** clause. If the operation contains a **sender** clause but no **from** clause, the sender shall be type compatible with the type of the variable or parameter referenced in the **sender** clause.

NOTE 3: An error due to a type mismatch may happen if the types in the receive part are not compatible to the types in the assignment part or, if the **from** clause is missing, but the type of the sender can be determined and it is not type compatible with the type in the **sender** clause.

m) When assigning implicitly decoded parameters (by using the **@decoded** modifier) in cases where the value or template to be matched uses the *MatchDecodedContent* (**decmatch**) matching for the parameter to be stored, the type of the template in the *MatchDecodedContent* matching shall be type-compatible to the type of the variable the decoded field is stored into.

Examples

EXAMPLE 1: Basic getcall

```
myPort.getcall(MyProc: s_myProcTemplate(5, v_myVar));   // accepts a call of MyProc at myPort

myPort.getcall(MyProc:{5, v_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 (v_myPar1Var, v_myPar2Var);
// The in or inout parameter values of MyProc are assigned to v_myPar1Var and v_myPar2Var.

myPort.getcall(MyProc:{5, v_myVar}) -> sender v_mySenderVar;
// Accepts a call of MyProc at myPort with the in or inout parameters 5 and v_myVar.
// The address of the calling party is retrieved and stored in v_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, v_myVar}) -> param(v_myVar1, v_myVar2) sender v_mySenderVar;

myPort.getcall(MyProc:{5, ?}) -> param(v_myVar1, v_myVar2);

myPort.getcall(MyProc:{?, v_myVar}) -> param( - , v_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 (v_myVarA, v_myVarB, - , -, v_myVarE);
// The parameters A, B, and E are assigned to the variables v_myVarA, v_myVarB, and
// v_myVarE. The out parameter D needs not to be considered.

myPort.getcall(MyProc2:{?, ?, 3, -, ?}) -> param (v_myVarA:= A, v_myVarB:= B, v_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 (v_myVarE:= E);
// Only the inout parameter value is needed for the further test case execution

// The following example demonstrates the use of encoded parameters:
signature MyProc3(in integer paramType, octetstring encodedParam);
template integer mw_int := ?;
…
var integer v_myVarX;
myPort.getcall(MyProc3:{1, decmatch mw_int}) -> param (v_myVarX := @decoded encodedParam);
// The parameters encodedParam is decoded into an integer and assigned to v_myVarX.
```

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 v_mySenderVar; // Removes a call from myPort and retrieves
                                        // the address of the calling entity
```

EXAMPLE 4: Getcall on any port

```
any port.getcall(MyProc:?)
```

EXAMPLE 5: Getcall on any port from port array

```
type port MyPort procedure { inout MyProc }
type component MyComponent {
  port MyPort p[10][10];
}
var integer v_i[2];
any from p.getcall(MyProc:?) -> @index value v_i;
// checking for an incoming call of the type MyProc on any port of the port array p and storing
// the index of the port on which the matching was successful first
```

22.3.3 The Reply operation

The **reply** operation is used to reply to a call.

Syntactical Structure

```
Port "." reply "(" TemplateInstance [ value TemplateBody ] ")"
[ 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 or template shall be explicitly stated with the **value** keyword and is first evaluated before returning.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5 and shown in table 15, 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) All *AddressRef* items in the **to** clause shall be of type **address**, **component** or of the address type bound to the port type (see clause 6.2.9) of the port instance referenced in the **reply** operation. No *AddressRef* in the **to** clause shall contain the special value **null** at the time of the operation.

e) If a value is to be returned to the calling party, this shall be explicitly stated using the **value** keyword. The *TemplateBody* in the **value** clause shall conform to the template(value) restriction.

f) Applying a **reply** operation to an unmapped or disconnected port shall cause a test case error.

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, v_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 | any from PortArrayRef ) "." getreply
[ "(" TemplateInstance [ value TemplateInstance ]")" ]
[ from Address ]
[ "->" [ value (VariableRef |
                ( "(" { VariableRef [ ":=" [ @decoded [ "(" Expression ")" ] ]
                                        FieldOrTypeReference ][","] } ")" )
              )]
       [ param "(" { ( VariableRef ":=" [ @decoded [ "(" Expression ")" ] ]
                                          ParameterIdentifier ) "," } |
                   { ( VariableRef | "-" ) "," }
              ")" ]
       [ sender VariableRef ]
       [ @index value VariableRef ] ]
```

NOTE 1: *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 followed by a specification of an address or component reference, a list of address or component references or **any component**..

NOTE 2: The one-to-one connection is considered to be a simple case of the one-to-many connections and allows the usage of the **from**-clause.

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.

When assigning individual parameters or referenced fields of the return value of a reply, encoded parameters can be decoded prior to assignment using the **@decoded** modifier. In this case, the referenced parameter or field of the return value on the right hand sided of the assignment shall be one of the **bitstring**, **hexstring**, **octetstring**, **charstring** or **universal charstring** types. It shall be decoded into a value of the same type as the variable on the left hand side of the assignment. Failure of this decoding shall cause a test case error. In case the parameter or referenced field of the return value is of the **universal charstring** type, the **@decoded** clause can contain an optional parameter defining the encoding format. The parameter shall be of the **charstring** type and it shall contain one of the strings allowed for the **decvalue_unichar** function (specified in clause C.5.4). Any other value shall cause an error. In case the parameter or referenced field of the return value is not a **universal char string**, the optional parameter shall not be present.

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.

Get a reply on any port from a port array

To get a reply on any port from a specific port array, use the **any from** *PortArrayRef* syntax where PortArrayRef shall be a reference to a port array identifier**.** It is also possible to store the index of a port in a single-dimensional port array at which the operation was successful to a variable of type integer or, in case of multi-dimensional port arrays the index of the successful port to an integer array or record of integer variable. When checking the port array for matching replies, the port indices to be checked are iterated from lowest to highest. If the port array is multi-dimensional, then the ports are iterated over from innermost to outermost array dimension from lowest to highest index for each dimension, e.g. [0][0], [0][1], [1][0], [1][1]. The first port which matches all the criteria will cause the operation to be successful even if other ports in the array would also meet the criteria.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5 and shown in table 15, 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) All *AddressRef* items in the **from** clause and all *VariableRef* items in the **sender** clause shall be of type **address**, **component** or of the address type bound to the port type (see clause 6.2.9) of the port instance referenced in the **getreply** operation. No *AddressRef* in the **from** clause shall contain the special value **null** at the time of the operation.

e) The *PortArrayRef* shall be a reference to a completely initialized port array .

f) The index redirection shall only be used when the operation is used on an any from port array construct.

g) If the index redirection is used for single-dimensional arrays, the type of the integer variable shall allow storing the highest index of the respective port array.

h) If the index redirection is used for multi-dimensional arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective port array, and the its type shall allow storing the highest index (from all dimensions) of the port array.

i) If a variable referenced in the **value**, **param**, **sender** or **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **getreply** operation, i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **getreply** operation.

j) If the **getreply** operation contains both **from** and **sender** clause, the variable or parameter referenced in the **sender** clause shall be type compatible with the template in the **from** clause. If the operation contains a **sender** clause but no **from** clause, the sender shall be type compatible with the variable or parameter referenced in the **sender** clause.

NOTE 3: An error due to a type mismatch may happen if the types in the receive part are not compatible to the types in the assignment part or, if the **from** clause is missing, but the type of the sender can be determined and it is not type compatible with the type in the **sender** clause.

k) When assigning implicitly decoded parameters or referenced fields of the return value (by using the **@decoded** modifier) in cases where the value or template to be matched uses the *MatchDecodedContent* (**decmatch**) matching for the parameter to be stored, the type of the template in the *MatchDecodedContent* matching shall be type-compatible to the type of the variable the decoded field is stored into.

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 v_myRetValue param(v_myPar1, v_myPar2);
// The returned value is assigned to variable v_myRetValue and the value
// of the two out or inout parameters are assigned to the variables v_myPar1 and v_myPar2.

myPort.getreply(MyProc1:{?, ?} value ?)-> value v_myRetValue param(- ,v_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(s_aTemplate) -> param( - , - , - , v_myVarOut1, v_myVarInout1);

myPort.getreply(s_aTemplate) -> param(v_myVarOut1:=D, v_myVarOut2:=E);

myPort.getreply(MyProc2:{ - , - , - , 3, ?}) -> param(v_myVarInout1:=E);

// The following example demonstrates the use of encoded parameters:
signature MyProc3(out integer paramType, out octetstring encodedParam);
template integer mw_int := ?;
…
var integer v_myVarX;
myPort.getreply(MyProc3:{1, decmatch mw_int}) -> param (v_myVarX := @decoded encodedParam);
// The parameters encodedParam is decoded into an integer and assigned to v_myVarX.
```

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 v_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:?)
```

EXAMPLE 5: Get a reply on any port from port array

```
type port MyPort procedure { inout MyProc }
type component MyComponent {
  port MyPort p[10][10];
}
var integer v_i[2];
any from p.getreply(MyProc:?) -> @index value v_i;
// Getting a reply of the type MyProc on any port of the port array p and
// storing the index of the port on which the matching was successful first
```

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 conforming to the template(value) restriction 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 and shown in table 15, 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) All *AddressRef* items in the **to** clause shall be of type **address**, **component** or of the address type bound to the port type (see clause 6.2.9) of the port instance referenced in the **raise** operation. No *AddressRef* in the **to** clause shall contain the special value **null** at the time of the operation.

e) Applying a **raise** operation to an unmapped or disconnected port shall cause a test case error.

f) The *TemplateInstance* shall conform to the template(value) restriction (see clause 15.8).

Examples

```
myPort.raise(MySignature, v_myVariable + v_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 | any from PortArrayRef ) "." catch
[ "(" ( Signature "," TemplateInstance ) | TimeoutKeyword ")" ]
[ from Address ]
[ "->" [ value ( VariableRef |
                 ( "(" { VariableRef [ ":=" [ @decoded [ "(" Expression ")" ] ]
                     FieldOrTypeReference ][","] } ")" )
               ) ]
       [ sender VariableRef ]
       [ @index value VariableRef ] ]
```

NOTE 1: *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 followed by a specification of an address or component reference, a list of address or component references or **any component**.

NOTE 2: The one-to-one connection is considered to be a simple case of the one-to-many connections and allows the usage of the **from**-clause.

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.

When assigning individual fields of an exception, encoded payload fields can be decoded prior to assignment using the **@decoded** modifier. In this case, the referenced field on the right hand sided of the assignment shall be one of the **bitstring**, **hexstring**, **octetstring**, **charstring** or **universal charstring** types. It shall be decoded into a value of the same type as the variable on the left hand side of the assignment. Failure of this decoding shall cause a test case error. In case the referenced field is of the **universal charstring** type, the **@decoded** clause can contain an optional parameter defining the encoding format. The parameter shall be of the **charstring** type and it shall contain one of the strings allowed for the **decvalue_unichar** function (specified in clause C.5.4). Any other value shall cause an error. In case the referenced field is not a **universal charstring**, the optional parameter shall not be present.

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.

Catch on any port from a port array

To **`catch`** an exception on any port from a specific port array, indices use the **`any from`** *`PortArrayRef`* syntax where PortArrayRef shall be a reference to a port array identifier**`.`** It is also possible to store the index of a port in a single-dimensional port array at which the operation was successful to a variable of type integer or, in case of multi-dimensional port arrays the index of the successful port to an integer array or record of integer variable. When checking the port array for matching exceptions, the port indices to be checked are iterated from lowest to highest. If the port array is multi-dimensional, then the ports are iterated over from innermost to outermost array dimension from lowest to highest index for each dimension, e.g. [0][0], [0][1], [1][0], [1][1]. The first port which matches all the criteria will cause the operation to be successful even if other ports in the array would also meet the criteria.

The catch on any port from a port array operation can not be used to catch a call timeout.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5 and shown in table 15, 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) All *AddressRef* items in the **from** clause and all *VariableRef* items in the **sender** clause shall be of type **`address`**, **`component`** or of the address type bound to the port type (see clause 6.2.9) of the port instance referenced in the **`catch`** operation. No *AddressRef* in the **`from`** clause shall contain the special value **`null`** at the time of the operation.

g) The *PortArrayRef* shall be a reference to a completely initialized port array.

h) The index redirection shall only be used when the operation is used on an any from port array construct.

i) If the index redirection is used for single-dimensional arrays, the type of the integer variable shall allow storing the highest index of the respective port array.

j) If the index redirection is used for multi-dimensional arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective port array, and the its type shall allow storing the highest index (from all dimensions) of the port array.

k) If a variable referenced in the **value**, **sender** or **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **catch** operation, i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **catch** operation.

l) If the **catch** operation contains both **from** and **sender** clause, the variable or parameter referenced in the **sender** clause shall be type compatible with the template in the **from** clause. If the operation contains a **sender** clause but no **from** clause, the sender shall be type compatible with the variable or parameter referenced in the **sender** clause.

NOTE 3: An error due to a type mismatch may happen if the types in the receive part are not compatible to the types in the assignment part or, if the from clause is missing, but the type of the sender can be determined and it is not type compatible with the type in the sender clause.

m) When assigning implicitly decoded exception fields (by using the **@decoded** modifier) in cases where the value or template to be matched uses the *MatchDecodedContent* (**decmatch**) matching for the parameter to be stored, the type of the template in the *MatchDecodedContent* matching shall be type-compatible to the type of the variable the decoded field is stored into.

Examples

EXAMPLE 1: Basic catch

```
myPort.catch(MyProc, integer: v_myVar);    // Catches an integer exception of value
                                           // v_myVar raised by MyProc at port myPort.

myPort.catch(MyProc, v_myVar);             // Is an alternative to the previous example.

myPort.catch(MyProc, v_a<v_b);             // Catches a boolean exception

myPort.catch(MyProc, MyType:{5, v_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 v_myVar;
// Catches an exception from myPartner and assigns its value to v_myVar.

myPort.catch(MyProc, s_myTemplate(5)) -> value v_myVarTwo sender myPeer;
// Catches an exception, assigns its value to v_myVarTwo and retrieves the
// address of the sender.

myPort.catch(MyProc, s_myTemplate(5)) -> value (v_myVarThree:= f1)
                                         sender myPeer;
// Catches an exception, assigns the value of its field f1 to v_myVarThree and retrieves the
// address of the sender.

// Handling encoded exception payload:

type MyException record {
  ...
}
type CommonException record {
  integer       exceptionId,
  octetstring   payload
}

signature S() exception (CommonException);
...

var MyException v_myVar;

myPort.catch (S, CommonException:{exceptionId := 25, payload := decmatch MyException:? })
                        -> value (v_myVar := @decoded payload);
// The encoded payload field of the caught exception is decoded and matched with m_excTemplate;
// if the matching is successful the decoded payload is stored in v_myVar.
```

EXAMPLE 3: The Timeout exception

```
myPort.call(MyProc:{5, v_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 v_mySenderVar;
```

EXAMPLE 5: Catch on any port

```
any port.catch;
```

EXAMPLE 6: Catch on any port from port array

```
type port MyPort procedure { inout MyProc }
type component MyComponent {
  port MyPort p[10][10];
}
var integer v_i[2];
any from p.catch(MyProc, MyType:?) -> @index value v_i;
// Catching an incoming exception of type MyType on any port in the port array p and
// storing the index of the port on which the matching was successful first
```

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 | any from PortArrayRef ) "." check
[ "("
     ( PortReceiveOp | PortGetCallOp | PortGetReplyOp | PortCatchOp ) |
     ( [ from Address ]
       [ "->" [ sender VariableRef ]
              [ @index value 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 from a specific sender

In the case of one-to-many connections the **check** operation may be restricted to a certain communication partner. This restriction shall be denoted using the **from** keyword followed by a specification of an address or component reference, a list of address or component references or **any component**.

NOTE 2: The one-to-one connection is considered to be a simple case of the one-to-many connections and allows the usage of the **from**-clause.

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.

Check on any port

To **check** on any port, use the **any port** keywords.

Check on any port from a port array

To **check** on any port from a specific port array, indicesindices use the **any from** *PortArrayRef* syntax where PortArrayRef shall be a reference to a port array identifier**.** It is also possible to store the index of a port in a single-dimensional port array at which the operation was successful to a variable of type integer or, in case of multi-dimensional port arrays the index of the successful port to an integer array or record of integer variable. When checking the port array for a matching message, call, reply or exception, the port indices to be checked are iterated from lowest to highest. If the port array is multi-dimensional, then the ports are iterated over from innermost to outermost array dimension from lowest to highest index for each dimension, e.g. [0][0], [0][1], [1][0], [1][1]. The first port which matches all the criteria will cause the operation to be successful even if other ports in the array would also meet the criteria.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5 and shown in table 15, 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) All *AddressRef* items in the **from** clause and all *VariableRef* items in the **sender** clause shall be of type **address**, **component** or of the address type bound to the port type (see clause 6.2.9) of the port instance referenced in the **check** operation. No *AddressRef* in the **from** clause shall contain the special value **null** at the time of the operation.

c) The *PortArrayRef* shall be a reference to a completely initialized port array.

d) The index redirection shall only be used when the operation is used on an any from port array construct.

e) If the index redirection is used for single-dimensional arrays, the type of the integer variable shall allow storing the highest index of the respective port array.

f) If the index redirection is used for multi-dimensional arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective port array, and the its type shall allow storing the highest index (from all dimensions) of the port array.

g) If a variable referenced in the **sender** or **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **check** operation, i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **check** operation.

h) If the **check** operation contains both **from** and **sender** clause, the variable or parameter referenced in the **sender** clause shall be type compatible with the template in the **from** clause. If the operation contains a **sender** clause but no **from** clause, the sender shall be type compatible with the variable or parameter referenced in the **sender** clause.

NOTE 3: In most cases the correct usage of the check operation can be checked statically, i.e. before/during compilation.

NOTE 4: An error due to a type mismatch may happen if the types in the receive part are not compatible to the types in the assignment part or, if the from clause is missing, but the type of the sender can be determined and it is not type compatible with the type in the sender clause.

Examples

EXAMPLE 1: Basic check

```
myPort1.check(receive(5));  // Checks for an integer message of value 5.

myPort1.check(receive(charstring:?) -> value v_myCharVar);
// Checks for a charstring message and stores the message if the message type is charstring

myPort2.check(getcall(MyProc:{5, v_myVar}) from myPartner);
// Checks for a call of MyProc at port myPort2 from myPartner

myPort2.check(getreply(MyProc:{5, v_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 v_myVar.

myPort2.check(catch(MyProc, s_myTemplate(5, v_myVar)));

myPort2.check(getreply(MyProc1:{?, v_myVar} value *)-> value v_myReturnValue param(v_myPar1,-));

myPort.check(getcall(MyProc:{5, v_myVar}) from myPartner -> param (v_myPar1Var, v_myPar2Var));

myPort.check(getcall(MyProc:{5, v_myVar}) -> sender v_mySenderVar);
```

EXAMPLE 2: Check any operation

```
myPort.check;

myPort.check(from myPartner);

myPort.check(-> sender v_mySenderVar);
```

EXAMPLE 3: Check on any port

```
any port.check;
```

EXAMPLE 4: Check on any port from port array

```
type port MyPort procedure { inout MyProc }
type component MyComponent {
  port MyPort p[10][10];
}
var integer v_i[2];
any from p.check(catch(MyProc, MyType:?)) -> @index value v_i;
// Checking for an incoming exception of the type MyType on any port of the port array p and
// storing the index of the port on which the matching was successful first
```

22.5 Controlling communication ports

22.5.0 General

TTCN-3 operations for controlling message-based and procedure-based ports are presented in table 24.

Table 24: 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**
Check the state of a port	**checkstate**

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 and shown in table 15.

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 and shown in table 15.

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 and shown in table 15.

Examples

```
myPort.receive (mw_myTemplate1) -> value v_recPDU;
                                   // the received value is decoded, matched against
                                   // MyTemplate1 and the matching value is stored
                                   // in the variable v_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 (mw_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 and shown in table 15.

Examples

```
myPort.halt;                          // No sending allowed on myPort from this moment on;
                                      // processing of messages in the queue still possible.
myPort.receive (mw_myTemplate1);      // If a message was already in the queue before the halt
                                      // operation and it matches mw_myTemplate1, it is processed;
                                      // otherwise the receive operation blocks.
```

22.5.5 The Checkstate port operation

The **checkstate** port operation allows to check the state of a port.

Syntactical Structure

```
( Port | ( all port ) | ( any port )) "." checkstate "(" SingleExpression ")"
```

Semantic Description

The **checkstate** port operation allows to examine the state of a port. If a port is in the state specified by the parameter, the **checkstate** operation returns the Boolean value **true**. If the port is not in the specified state, the **checkstate** operation returns the Boolean value **false**. Calling the **checkstate** operation with an invalid argument leads to an error.

The checkstate operation allows to check for different dimensions of a port state. It allows to check if a port is Started, Halted or Stopped, but also if a port is Connected, Mapped or Linked (i.e. Connected or Mapped).

NOTE 1: The states Started, Halted and Stopped refer to the port states defined in the clauses F.3.1 and F.3.2. The states Connected, Mapped and Linked are related to the application of the connection operations **connect**, **disconnect**, **map** and **unmap** as defined in clause 21.1.

The **checkstate** port operation can be used with **all port** and **any port**. Using the **checkstate** operation with **any port** allows to test if at least one port of a test component is in the specified state. Using the **checkstate** operation with **all port** allows to check if all ports of a component are in the specified state.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5 and shown in table 15, the following restrictions apply:

a) The parameter of the **checkstate** operation shall be of type **charstring** and shall have one of the following values:

   a) "Started"

   b) "Halted"

   c) "Stopped"

   d) "Connected"

   e) "Mapped"

   f) "Linked"

NOTE 2: Clause E.2.2.4 includes the type definition `objState` and the constant definitions `STARTED`, `HALTED`, `STOPPED`, `CONNECTED`, `MAPPED`, and `LINKED`. It is recommended to use the **checkstate** operation in combination with this type and these constants to ease the checking of correct usage and to improve the readability of test specs.

b) Calling the **checkstate** operation with a **charstring** parameter not listed in a) shall lead to an error.

Examples

```
type component MyMTCType // Component type definition for an MTC
{
  port MyPortType pCO1, pCO2
}

type component MyTestSystemInterface // Component type definition for a test system interface
{
    port MyPortType pCO3, pCO4, pCO5;
}

// Test case definition
testcase TC_MyTestcase1 () runs on MyMTCType system MyTestSystemInterface {

    var boolean v_myPortState;

    myPortState := all port.checkstate("Started");  // checkstate returns true, because all
                                                    // ports of a component are started after
                                                    // component creation and start

    v_myPortState := any port.checkstate("Linked"); // checkstate returns false, no port is
                                                    // either connected nor mapped

    map(mtc:pCO1, system:pCO3);

    v_myPortState := pCO1.checkstate("Linked");     // checkstate returns true, pCO1 is mapped
    v_myPortState := pCO1.checkstate("Mapped");     // checkstate returns true, pCO1 is mapped

    v_myPortState := pCO1.checkstate("Connected");  // checkstate returns false, pCO1 is mapped
                                                    // and not connected

    v_myPortState := any port.checkstate("Mapped"); // checkstate returns true, pCO1 is mapped

    all port.stop;

    v_myPortState := all port.checkstate("Started");// checkstate returns false, all ports
                                                    // are stopped

    v_myPortState := pCO1.checkstate("Stopped");    // checkstate returns true, pCO1 is stopped


    // further testcase behaviour
    // …
}
```

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 25.

Table 25: 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**
checkstate	yes	yes	**any port.checkstate("Started")** **all port.checkstate("Connected")**

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

23.0 General

TTCN-3 supports a number of timer operations as given in table 26. These operations may be used in test cases, functions, altsteps and module control.

Table 26: 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 and shown in table 15, the following restrictions apply:

a) Timer value shall be a non-negative numerical **float** number (i.e. the value shall be greater than or equal to 0.0; infinity and not_a_number are disallowed).

Examples

```
t_myTimer1.start;           // t_myTimer1 is started with the default duration
t_myTimer2.start(20E-3);    // t_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 v_i := 0; v_i<=4; v_i:=v_i+1)
  { v_timerValues [v_i] := 1.0 }

for (var integer v_i := 0; v_i<=4; v_i:=v_i+1)
  {t_myTimer [v_i].start ( v_timerValues [v_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 and shown in table 15.

Examples

```
t_myTimer1.stop;    // stops t_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 and shown in table 15.

Examples

```
var float v_myVar;
v_myVar := t_myTimer1.read; // assign to v_myVar the time that has elapsed since t_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 |
      any from TimerArrayRef )
"." running
["->" @index value VariableRef ]
```

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.

When the **any from** *TimerArrayRef* notation is used, where TimerArrayRef shall be a timer array identifier, the timers from the referenced array are iterated over and their states are checked individually, from innermost to outermost dimension from lowest to highest index for each dimension. The first timer to be found in the running state causes the operation returning with the **true** value. If no running timer is found in the array, the operation returns with the **false** value. The index of the first timer found running can optionally be stored in an integer variable for a single-dimensional array, or to an integer array or record of integer variable for multi-dimensional timer arrays.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5 and shown in table 15, the following restrictions apply:

a) *TimerArrayRef* shall be a reference to a completely initialized timer array.

b) The index redirection shall only be used for **any from** timer array running operations.

c) If the index redirection is used for single-dimensional timer arrays, the type of the integer variable shall allow storing the highest index of the respective timer array.

d) If the index redirection is used for multi-dimensional timer arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective timer array, and its type shall allow storing the highest index (from all dimensions) of the timer array.

Examples

EXAMPLE 1: Checking if a specific timer is running

```
if (t_myTimer1.running) { … }
```

EXAMPLE 2: Checking if an arbitrary timer is running

```
if (any timer.running) { … }
```

EXAMPLE 3: Checking if an arbitrary timer from a timer array is running

```
timer t_myTimerArray[2][2];
var integer v_i[2];
if (any from t_myTimerArray.running -> @index value v_i;) { … }
// checks if any timer from array is running
// assigns index of matched timer to v_i
```

23.6 The Timeout operation

The **timeout** operation allows to check the expiration of timers.

Syntactical Structure

```
( ( ( TimerIdentifier | TimerParIdentifier ) { "[" SingleExpression "]" } ) |
      any timer |
      any from TimerArrayRef )
"." timeout
["->" @index value VariableRef ]
```

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. In this case a randomly chosen timeout event is removed from the timeout-list.

When the **any from** *TimerArrayRef* notation is used, where TimerArrayRef shall be a timer array identifier, the timers from the referenced array are iterated over and individually checked for timeout from innermost to outermost dimension from lowest to highest index for each dimension. The first timer to be found in the timeout-list causes that timer to be removed from the list and the timeout operation succeeds. The index of the matched timer can be optionally stored in an integer variable for single-dimensional arrays or to an integer array or record of integer variable for multi-dimensional timer arrays.

Restrictions

In addition to the general static rules of TTCN-3 given in clause 5 and shown in table 15, the following restrictions apply:

a) The **`timeout`** operation does not return any value and therefore shall not be used in an expression.

b) *TimerArrayRef* shall be a reference to a completely initialized timer array.

c) The index redirection shall only be used for **any from** timer array timeout operations.

d) If the index redirection is used for single-dimensional timer arrays, the type of the integer variable shall allow storing the highest index of the respective timer array.

e) If the index redirection is used for multi-dimensional timer arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective timer array, and its type shall allow storing the highest index (from all dimensions) of the timer array.

Examples

EXAMPLE 1: Timeout of a specific timer

```
t_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
```

EXAMPLE 3: Timeout of a timer from a timer array

```
timer t_myTimerArray[2][2];
var integer v_i[2];
any from t_myTimerArray.timeout -> @index value v_i;
// checks for the timeout of any timer from array
// assigns index of matched timer to v_i
```

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 27.

Table 27: 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

24.0 General

Verdict operations given in table 28 allow to set and retrieve verdicts. These operations shall only be used in test cases, altsteps and functions.

Table 28: 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.

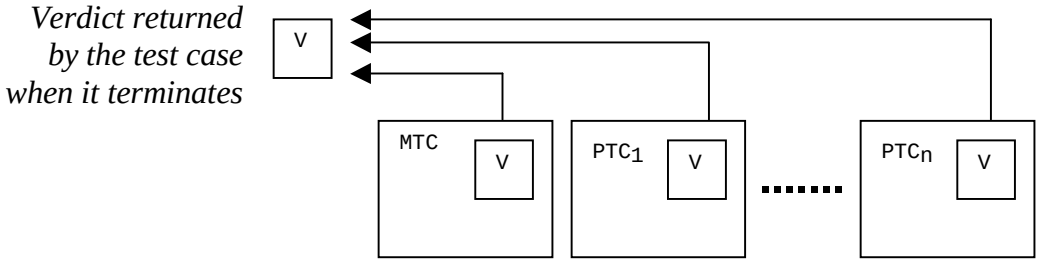


Figure 14: Illustration of the relationship between verdicts

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 29. 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 29.

Table 29: 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. runtime) 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 29.

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 and shown in table 15, 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 runtime 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 17 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 v_myVar:= 1;
:
myPort.receive(integer:v_myVar);// Matches an integer value with the value of v_myVar
                                // at port myPort
setverdict(pass, "Value received: ", v_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 and shown in table 15.

Examples

```
v_myResult := getverdict; // Where v_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 and shown in table 15, the following restrictions apply:

a) *Expression* shall have the base type charstring or universal charstring.

Examples

```
var charstring v_myString:= " now."
action("Send MyTemplate on lower PCO" & v_myString);    // Informal description of the
                                                        // external action
```

26 Module control

26.0 General

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 30.

Table 30: 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**
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 and shown in table 15, 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 test behaviour chain called from a test case, i.e. test cases can only be executed from the control part or from functions or altsteps called directly or indirectly 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(TC_MyTestCase1());  // executes TC_MyTestCase1, without storing the
                            // returned test verdict and without time supervision
```

EXAMPLE 2: Test case execution with keeping the test case verdict

```
v_myVerdict := execute(TC_MyTestCase2());   // executes TC_MyTestCase2 and stores the resulting
                                            // verdict in variable v_myVerdict
```

EXAMPLE 3: Test case timer

```
v_myVerdict := execute(TC_MyTestCase3(),5E-3);
// executes TC_MyTestCase3 and stores the resulting verdict in variable v_myVerdict.
// If the test case does not terminate within 5ms, v_myVerdict will get the value 'error'
```

EXAMPLE 4: Host id

```
v_myVerdict := execute(TC_MyTestCase3(), -, "Host1");
// executes TC_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 and shown in table 15, 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 invoked by module control.

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 15.

Examples

EXAMPLE 1: Test case execution in a loop

```
module MyTestSuite () {
     :
    control {
         :
        // Do this test 10 times
        v_count:=0;
        while (v_count < 10)
        {   execute (TC_MySimpleTestCase1());
            V_count := v_count+1;
        }
    }
}
```

EXAMPLE 2: Test case execution controlled by a timer and a counter

```
// Example of the use of the running timer operation
while (t_t1.running or v_x<10)  // Where t_t1 is a previously started timer
{   execute(TC_MyTestCase());
    v_x := v_x+1;
}

// Example of the use of the start and timeout operations

timer t_t1:= 1.0;
 :
execute(TC_MyTestCase1());
t_t1.start;
t_t1.timeout;   // Pause before executing the next test case
execute(TC_MyTestCase2());
```

EXAMPLE 3: Selection/deselection of test cases with Boolean expressions

```
module MyTestSuite () {
     :
    control {
         :
        if (f_mySelectionExpression1()) {
            execute(TC_MySimpleTestCase1());
            execute(TC_MySimpleTestCase2());
            execute(TC_MySimpleTestCase3());
        }
        if (f_mySelectionExpression2()) {
            execute(TC_MySimpleTestCase4());
            execute(TC_MySimpleTestCase5());
            execute(TC_MySimpleTestCase6());
        }
         :
    }
}
```

EXAMPLE 4: Selection/deselection of test cases with functions

```
function f_myTestCaseGroup1()
{   execute(TC_MySimpleTestCase1());
    execute(TC_MySimpleTestCase2());
    execute(TC_MySimpleTestCase3());
}
function f_myTestCaseGroup2()
{   execute(TC_MySimpleTestCase4());
    execute(TC_MySimpleTestCase5());
    execute(TC_MySimpleTestCase6());
}
 :
control
{   if (f_mySelectionExpression1()) { f_myTestCaseGroup1(); }
    if (f_mySelectionExpression2()) { f_myTestCaseGroup2(); }
     :
}
```

EXAMPLE 5: Selection/deselection of test cases based on test case verdicts

```
if ( execute (TC_MySimpleTestCase()) == pass )
   { execute (TC_MyGoOnTestCase()) }
else
   { execute (TC_MyErrorRecoveryTestCase()) };
```

27 Specifying attributes

27.0 General

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

27.1.0 General

Attributes can be associated with TTCN-3 language elements by means of the with statement. The with statement can be applied to modules, global module definitions and to local definitions in control, test cases, functions, altsteps, statement blocks and in component type definitions.

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) or to members of component or port types, 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.

A **with** statement can follow any module, any global definition inside module and group declarations as well as any local definition in component types and statement blocks inside behaviour definitions or the control part.

Attibutes can be attached to synonym types (6.4). If the synonym type is a structured type, attributes in the **with** statement may reference fields or elements of this structured type.

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 green
```

27.1.2 Overwriting rules for attributes

27.1.2.0 General

An attribute definition that is directly attached to a lower scope unit will override a general attribute definition in a higher scope and a type-specific attribute inherited from a type reference. Attributes inherited from a type reference will override general attributes from a higher scope unit containing the type reference. Additional overwriting rules for variant attributes are defined in clause 27.1.2.1.

EXAMPLE 1:

```
type record MyRecordA
{
    :
} with { encode "RuleA" }

// In the following, MyRecordA is encoded according to "RuleA" and not according to
// "RuleB" because the attribute from the referenced type MyRecordA overrides
// the attribute from higher scope unit (surrounding MyRecordB type).

type record MyRecordB
{
     :
    MyRecordA   field
} 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. If multiple attributes of the same type are allowed, all of them are overridden unless specified otherwise.

EXAMPLE 2:

```
// 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" }
    }
}
```

Attributes defined for a synonym type don't override existing attributes of fields or elements of this synonym type. The attributes are applied to the fields or elements of synonym types only if the fields or elements have no valid attibutes.

EXAMPLE 3:

```
// Example of the use of attributes in synonym types
type record SourceType1 {
    integer field1,
    integer field2
} // neither the record nor its fields have a valid attribute

type record SourceType2 {
    integer field1,
    integer field2
} with { encode  "Rule1" }
// the record and its fields have a valid encode attribute "Rule1"

type record SourceType1 SynonymType1 with { encode  "Rule2" }
// SynonymType1 and all its fields will be encoded with Rule2

type record SourceType2 SynonymType2 with { encode  "Rule3" }
// SynonymType2 will be encoded with Rule3, but field1 and field2 will be encoded with
// Rule1 as SourceType2 definition already specifies the encode attribute of these fields
```

Attributes with the **@local** modifier **o**verride attributes from higher scope, but they are valid for the associated language element only. They do not affect definitions inside the associated language element as the **@local** attribute is completely transparent to lower scopes. Attributes from higher scope will still affect attributes in lower scopes even if the **@local** attribute is between them.

NOTE: Attributes with the **@local** modifier associated to modules and groups are valid, but do not affect the definitions inside them.

EXAMPLE 4:

```
module M {
    type record MyRec {
        integer field1,
        integer field1,
    } with { encode @local "CodecB" }
    // the record type MyRec will be encoded with CodecB, but its fields with CodecA,
    // because the local attribute CodecB doesn't affect fields of the MyRec type.
} with { encode "CodecA" }
```

An attribute definition in a lower scope or those inherited from a referenced type can be overwritten in a higher scope by using the **`override`** directive.

EXAMPLE 5:

```
type record MyRecordA
{
    :
} with { encode "RuleA" }

// In the following, fieldA of a MyRecordB instance is encoded according to RuleB
type record MyRecordB
{
    :
    MyRecordA   fieldA
} with { encode override (fieldA) "RuleB" }
```

The **`override`** directive overrides the specified attribute for all declarations at all lower scopes that do not also declare the specified attribute. If the override directive is applied to a type reference, it doesn't affect the attributes of the original referenced type.

An attribute definition directly attached to a field or element of a structured type overrides the corresponding attribute of the structured type, as regards the identified field or element. Override attribute applied to a synonym type (clause 6.4) overrides attributes of all fields or elements of the synonym type unless the synonym type definition contains an explicit attribute definition for the field or element.

EXAMPLE 6:

```
// An instance of MyRecordA is encoded according to RuleA.
type record MyRecordA
{
    :
} with { encode override "RuleA" }

// In the following, fieldA of a MyRecordB instance (and all its sub-fields) is encoded
// according to "RuleB".
type record MyRecordB
{
    :
    MyRecordA   fieldA
} with { encode override "RuleB" }

// The following template will use "RuleA" as the override directive for MyRecordB affects only
// MyRecordB.fieldA, but not the original MyRecordA.
template MyRecordA mw_msg;

// In the following, rule "RuleB" is overridden by "RuleC" for fieldC, but it is
// not overridden by "RuleA" of the group because the direct attachment to fieldC and
// MyRecordC override the encode of the outer scope.
group myGroup {
  type record MyRecordC
  {
      :
  } with { encode override "RuleB" }

  type record MyRecordD
  {
      :
      MyRecordC     fieldC
 } with { encode override (fieldC) "RuleC" }
} with { encode override "RuleA" }

// In the following, the template mw_msg will be encoded with "RuleB", because the
// override directive doesn't override the encode attribute in references. However,
```

```
// all fields of the mw_msg_template will be encoded with "RuleA", because the attributes
// from the references have higher precedence than attributes from a higher scope.
type record MyRecordE
{
  :
} with { encode override "RuleA" }

template MyRecordE mw_msg :=
{
  :
} with { encode "RuleB" }

// MyRecordG and its "field1" member will be encoded with "RuleB", but its field2 member
// will be encoded with "RuleA", because there's an encode attribute explicitly declared
// for this field.
type record MyRecordF {
    integer field1,
    integer field2
} with { encode "RuleA" }

type MyRecordF MyRecordG with {
    encode override "RuleB";
    encode(field2) "RuleA"
}
```

27.1.2.1 Additional default 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.

The present document defines the default rules for variant attributes. Extension packages of TTCN-3, for example specifying language mappings, may define their own overwriting rules for variant attributes. For variant attributes the following default overwriting rules apply:

- a **variant** attribute overwrites a 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 MyType; // 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 { encode (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 1"
                                // using encoding variant "Variant 2"
        }
        with { variant (field1) "Variant 3" }
        :
    }
    with { encode "Encoding 3"; variant "Variant 2"}

}
with { encode "Encoding 1" }
```

27.1.2.2 Overwriting rules for multiple encoding

Explicitly listed encode attributes that occur on the higher scope and are not overwritten will retain all variants related to them.

An encoding related variant will overwrite only variants related to the same encoding.

EXAMPLE:

```
type integer Int with {
    encode "CodecA"; variant "CodecA"."Rule1";
    encode "CodecB"; variant "CodecB"."Rule2";
}

// Modifying list of allowed encodings
type Int Int2 with {
    encode "CodecA"; // variant "CodecA"."Rule1" is kept
    encode "CodecC"; variant "CodecC"."Rule6"; // new encoding and related variant
    // "CodecB" encoding together with its variant are discarded as "CodecB" is not
    // explicitly referenced
}

// Overwriting variant with an encoding reference
type Int Int3 with {
    variant "CodecB"."Rule4"; // new variant for encoding "CodecB" overwrites
        // the original variant "CodecB"."Rule2"
        // Variant "CodecA"."Rule1" is unchanged as this definition contains no reference
        // to "CodecB"
}
```

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.

When resolving the attributes, the **`import`** statement works as an additional higher scope unit on the top of the imported module. Attributes set in the import statement are valid only within the importing module.

NOTE 1: The import statement occurs inside an importing module and sometimes inside a group. Because of the scope rules, attributes of these scope units apply to the imported module too.

NOTE 2: If a **`with`** statement is added to an import of a definition where a local definition also has a **`with`** statement, the local definition's attributes overwrite the attributes added to the import statement in the normal way. Thus, if the attributes of a local definition shall be changed via the import statement, the override directive needs to be used.

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 | @local ]
    ["(" DefinitionRef | FieldReference | AllRef ")"]
    [ ( FreeText | ("{" FreeText { "," FreeText } "}") ) "." ] 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.

DefinitionRef and *FieldReference* identify a definition or field respectively which is within the module, group or definition to which the **`with`** statement is associated.

AllRef can be used to apply attributes to multiple language elements defined within the scope to which the **`with`** statement is associated. *AllRef* provides a flexible mechanism to select all language elements or all language elements of a certain kind defined in a given scope. Individual language elements that are not affected by an attribute can be excluded from a set of selected language elements in the **`except`** clause.

Restrictions

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

a) *DefinitionRef* and *FieldReference* shall refer to a definition or field respectively which is within the module, group or definition to which the with statement is associated.

b) In case multiple attributes of the same type are allowed, all of them shall be without an additional modifier (**`override`**, **`@local`**) or the modifier shall be the same for all attributes.

c) Dot notation in the *FreeText* part is allowed for variant attributes only.

EXAMPLE:

```
type record MyService {
    integer i,
    float f
}
with { display "ServiceCall" }      // MyRecord will be displayed as a ServiceCall

group G {
    ...
} with { encode(template all except (mw_msg1)) "Rule1" }
// with the exception of mw_msg1, all templates defined in this group will be encoded
// using the "Rule1" encoding
```

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 graphical presentation format can be found in ETSI 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

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

a) At most one display attribute shall be applied to each definition, each individual field reference or language element to which a **with** statement is associated.

EXAMPLE:

```
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`**.

The **`with`** statement may contain more than one **`encode`** attribute. In this case, multiple encodings are supported in the context where the attribute is used. The encoding used in the encoding and decoding operations can be selected dynamically by using the **`setencode`** operation (clause 27.9), as a parameter of predefined codec functions (clause C.5) or inside the codec implementation.

Restrictions

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

EXAMPLE:

```
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 E.2.1):

a) "`8 bit`" and "`unsigned 8 bit`" mean, when applied to integer and enumerated types, that the integer value or the integer numbers associated with enumerated values 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 enumerated values 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 enumerated values 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 enumerated values 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 E.2.2):

a) "`UTF-8`" means, when applied to the universal charstring type, that the value shall be encoded and decoded according to the UCS encoding scheme UTF-8 as defined in clause 10.1 of ISO/IEC 10646 [2].

b) "`UTF-16`" means, when applied to the universal charstring type, that the value shall be encoded and decoded according to the UCS encoding scheme UTF-16 as defined in clause 10.4 of ISO/IEC 10646 [2].

c) "`UTF-16LE`" means, when applied to the universal charstring type, that each character of the value shall be individually encoded and decoded according to the UCS Encoding scheme UTF-16LE as defined in clause 10.3 of ISO/IEC 10646 [2].

d) "`UTF-16BE`" means, when applied to the universal charstring type, that the value shall be encoded and decoded according to the UCS Encoding scheme UTF-16BE as defined in clause 10.2 of ISO/IEC 10646 [2].

e) "UTF-32" means, when applied to the universal charstring type, that the value shall be encoded and decoded according to the UCS Encoding scheme UTF-32 as defined in clause 10.7 of ISO/IEC 10646 [2].

f) "UTF-32LE" means, when applied to the universal charstring type, that the value shall be encoded and decoded according to the UCS Encoding scheme UTF-32LE as defined in clause 10.6 of ISO/IEC 10646 [2].

g) "UTF-32BE" means, when applied to the universal charstring type, that the value shall be encoded and decoded according to the UCS Encoding scheme UTF-32BE as defined in clause 10.5 of ISO/IEC 10646 [2].

h) "`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).

NOTE: The UCS Encoding schemes allow an optional signature (also known as byte order mark, BOM) to be present in encoded character strings. The above UCS encoding scheme variant attributes does not specify, if signatures are present in the encoded values or not, this is an option for the encoder. It is expected that decoders are able to process signatures in the decoding process.

The following strings are the predefined (standardized) **`variant`** attributes for structured types (see clause E.2.2.4):

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 ETSI 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.

Multiple encodings

If multiple encodings (see clause 27.4) are used, the **`variant`** attribute value shall be composed of two parts separated by a dot. Such variant attributes are called encoding related variant attributes. The first part of the attribute specifies the encodings the variant is related to. There are two possible notations: either a simple string when the variant is related to a single **`encode`** attribute or a comma separated list of strings enclosed in curly brackets if the variant is related to multiple encodings. The second part of the attribute (following the dot symbol) is a simple string that specifies the variant value.

The encoding related attributes are valid only when the related encoding is selected.

It is not allowed to define **`variant`** attributes with no encoding reference if multiple encodings are used.

Multiple variants

The **`with`** statement can contain any number of variant attributes.

Restrictions

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

a) When dot notation is used in the variant attribute value for an element, the strings preceding the dot symbol shall resolve into one of the encode attribute values associated with the same element.

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" }

    type charstring Multi with {
        encode "Codec1"; variant "Codec1"."Rule1";
        encode "Codec2"; variant "Codec2"."Rule3";
    };  // multiple encodings ("Codec1", "Codec2"), the variant "Rule1" is valid
        // for the "Codec1" encoding only, while the variant "Rule3" applies only
        // for the "Codec2" encoding

    type charstring Multi2 with {
        encode "Codec1";  encode "Codec2";
        variant {"Codec1","Codec2"}."Rule1";
    };  // multiple encodings ("Codec1", "Codec2"), variant "Rule1" applies to both of them

    type charstring Multi3 with {
        encode "Codec1";  encode "Codec2";
        variant "Rule1";
    }   // the statement will produce an error as there are multiple encodings and the
        // variant attribute doesn't specify encoding reference
```

```
}
with { encode "Global encoding rule" }
```

27.6 Extension attributes

Extension attributes can be used for proprietary extensions to TTCN-3. The **with** statement may contain any number of extension attributes.

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 TC_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, templates, variables and template variables of record and set types are implicitly set to **omit**.

Syntactical Structure

```
optional
```

Semantic Description

TTCN-3 constants, module parameters, templates, variables and template variables 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, templates, variables and template variables 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, that have no assigned value definition in the statement on which the attribute operates, 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, that have no assigned value definition in the statement on which the attribute operates, are left undefined. This applies recursively to the optional fields of the entity and to subfields of the mandatory fields.

For variables and template variables associated with an "`implicit omit`" optional attribute, recursive procedure is applied to their optional fields after each assignment or usage as **`out`** or **`inout`** actual parameter in the scope of their declaration (e.g.after re-assinging parts or all of a variable's value) setting all optional fields that have no assigned value definition to **`omit`**.

NOTE: Assinging the "`implicit omit`" attribute to a variable can have a negative runtime performance impact. Tool vendors are encouraged to identify and optimize particular cases where these operations are not needed (e.g. where it is possible to decide statically that no optional fields of the structure could have become undefined).

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 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 or import statement.

b) At most one optional attribute shall be applied to each definition, each individual field reference or language element to which a **`with`** statement is associated.

EXAMPLE:

```
type record MyRecord1 {
  integer a,
  boolean b optional
}
type record MyRecord2 {
  MyRecord1 m
}

// reference templates with explicitly set fields
template MyRecord1 mw_myTemplate1 := { a := ?, b := omit }
template MyRecord2 mw_myTemplate2 := { m := { a := ?, b := omit }}

// reference templates
template MyRecord1 mw_myTemplate1a := {a := ? } // b is undefined
template MyRecord1 mw_myTemplate1b := {a := ? } with {optional "explicit omit"} // b is
undefined

template MyRecord2 mw_myTemplate2a := {} // m and its subfields are undefined
template MyRecord2 mw_myTemplate2b := { m := { a := ?}}; // m.b is undefined

// templates with attribute

template MyRecord1 mw_myTemplate11 := { a := ? } with {optional "implicit omit"}
  // same as mw_myTemplate1, b is set to omit


template MyRecord2 mw_myTemplate21 := { m := { a := ?}} with {optional "implicit omit"}
  // same as mw_myTemplate2, by recursive application of the attribute
template MyRecord2 mw_myTemplate22 := { m := mw_myTemplate1a } with {optional "implicit omit"}
  // same as mw_myTemplate2, by recursive application of the attribute

template MyRecord2 mw_myTemplate23 := {} with {optional "implicit omit"}
  // same as mw_myTemplate2a, m remains undefined

template MyRecord2 mw_myTemplate24 := { m := mw_myTemplate1b } with {optional "implicit omit"}
  // same as mw_myTemplate2b, the attribute on the lower scope is not overwritten
template MyRecord2 mw_myTemplate25 := { m := MyTemplate1b }
  with {optional override "implicit omit"}
  // same as mw_myTemplate2, the attribute on the lower scope is overwritten

  // implicitly omitted fields stay omitted after assignment
template MyRecord1 mw_myTemplate3a := mw_myTemplate1a with {optional "implicit omit"}
  // same as mw_myTemplate1, b is set to omit
template MyRecord1 mw_myTemplate3b := mw_myTemplate3a;
  // same as mw_myTemplate1, b is set to omit, by implicit omit attribute of mw_myTemplate3a
template MyRecord1 mw_myTemplate3c := mw_myTemplate3a with {optional "explicit omit"}
  // same as mw_myTemplate1, b is set to omit, by implicit omit attribute of mw_myTemplate3a

  // implicitly omitted fields stay omitted after assignment
```

```
template MyRecord1 mw_myTemplate3a := mw_myTemplate1a with {optional "implicit omit"}
  // same as mw_myTemplate1, b is set to omit
template MyRecord1 mw_myTemplate3b := mw_myTemplate3a;
  // same as mw_myTemplate1, b is set to omit, by implicit omit attribute of mw_myTemplate3a
template MyRecord1 mw_myTemplate3c := mw_myTemplate3a with {optional "explicit omit"}
  // same as mw_myTemplate1, b is set to omit, by implicit omit attribute of mw_myTemplate3a

function f_helper1() return MyRecord1 {
  var MyRecord1 v_temp := { 1, true };
  return v_temp;
}

function f_helper2(out MyRecord1 p_par) {
  p_par := { 1 };
  // p_par is { 1, <undefined> }, no implicit omit attribute is in effect here
  ...
}

function f_function() {
  var MyRecord2 v_var1;
  v_var1.m.a := 5;
  // at this time v_var1.m.b is set to omit for the "implicit omit" attribute

  v_var1.m := f_helper1();
  // v_var1.m.b is true, checking of v_var1 might be skipped given strong static checks

  f_helper2( v_var1.m );
  // at this time v_var1.m.b is set to omit for the "implicit omit" attribute
  ...
} with {optional "implicit omit"}
```

27.8 Retrieving attribute values

TTCN-3 provides a set of operations that can be used for retrieving attribute values associated with a type, template, variable, constant or module parameter.

Syntactical Structure

```
( Type | TemplateInstance ) "." ( display | encode | variant | extension | optional )
[ "(" Expression ")" ]
```

Semantic Description

The operation returns the actual value of an attribute associated with the type, template, variable, constant or module parameter that precedes the dot symbol. The value preceding the dot symbol may be unitialized. The attribute kind is denoted by the keyword following the dot symbol.

The return value of the operations for retrieving attribute values is of a **`universal charstring`** type in case of attributes that can be present only once (**`display`**, **`optional`**) . If such an attribute is not defined, the operation returns an empty string. If the attribute can occur multiple times (**`encode`**, **`variant`**, **`extension`**), the operation returns a **`record of universal charstring`** type. If such an attribute is not present, the operation returns an empty record of value.

The operation for getting a **`variant`** attribute value may be followed by an optional parameter. If no parameter is present, the operation returns only variants that are not bound to any particular encoding. If the parameter is present, the returned value will containt variants that are bound to the encoding referenced by the parameter.

Restrictions

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

a) The optional parameter of the operation shall be used only for getting variant attributes.

b) The *Expression* in optional parameter of the operation shall be of the **`universal charstring`** type.

c) An error shall be produced if the *Expression* in the optional parameters is not one of the valid encode attributes.

EXAMPLE:

```
// MyPDU1 will be displayed as PDU
type record MyPDU1 { ... } with {
display "blue";
    variant "CommonRule";
    encode "Codec1";
    variant "Codec1"."Rule1A";
    variant "Codec1"."Rule1B";
    encode "Codec2";
    variant "Codec2"."Rule2A";
    variant "Codec2"."Rule2B";
}
type record of universal charstring RUC;
control {
    var MyPDU1 v_pdu;
    var universal charstring v_display;
    var RUC v_encoding, v_variants;
    v_display := MyPDU1.display;        // v_display will contain "blue"
    v_display := v_variants.display;    // v_display will contain "" as no display attribute is
                                        // defined for v_variants
    v_encoding := v_pdu.encode;         // v_encoding will contain { "Codec1", "Codec2" }
    v_variants := v_pdu.variant;        // v_variants will contain { "CommonRule" }

    // retrieve variants for all defined encodings
    for (var integer i := 0; i < sizeof(v_encoding); i := i + 1) {
        v_variants := v_pdu.variant(v_encoding[i]);
        ...
    }
    v_variants := v_variants.encode; // v_variants will contain {} as no encode attribute is
                                     // defined for v_variants
    v_variants := v_pdu.variant("UnknownCodec");    // produces an error as there is no such
                                                    // encode attribute as "UnknownCodec"
}
```

27.9 Dynamic configuration of encoding used by ports

The **setencode** operation can be used on a port or set of ports to dynamically select for the affected ports a single encode attribute value to be used for a type that has multiple encode attributes attached to it.

Syntactical Structure

```
( Port | ( all port ) | self ) "." setencode"(" Type "," SingleExpression ")"
optional
```

Semantic Description

The **setencode** operation dynamically restricts the number of encode attribute values of a referenced type or its fields or elements to a single value. Dependent on the language element preceding the dot, the encoding configuration is valid either for all sending and receiving operations of a single port (single port reference), sending and receiving operations of all ports of the current component (**all port** notation) or for all codec function and communication operation of the current component (**self** keyword).

If the referenced type contains multiple encode attributes and the expression provided in the **setencode** operation is equal to one of these encode attribute values, the statement reduces the list of encode attributes to the selected one. The procedure is applied recursively to all elements and fields or the referenced type. After executing the operation, all other encode attributes and variants related to them are dynamically disabled and invisible to the codec.

Repeated call of the **setencode** operation always uses the static attributes that are valid for the referenced type. Previous calls of the **setencode** operation referencing the type are not considered in this case. This way it is possible to change the encoding during test execution using different encodings.

It is allowed to reference a field or element of a type using an extended type reference in the **setencode** operation. This operation is useful for payload fields of container protocols and allows dynamic configuration of the proper encoding for payload fields. If the extended type reference is used, following calls of the **setencode** operation for the whole type or any element that contains the the referenced payload field won't change the encoding that was dynamically configured for this field or element (and its sub-fields).

Restrictions

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

a) If the **setencode** operation is applied to a single port, the referenced type shall be either listed in the **in** or **out** type list of the related port type or it shall be a reference to a field or element on any level of nesting of a type listed in the **in** or **out** type list of the related port type.

b) The *SingleExpression* used in the second parameter of the **setencode** operation shall be compatible with the **universal charstring** type.

EXAMPLE:

```
type port P message {
  inout PDU;
}

type component C {
port P p;
}

// Payload type with two encoding options
type record Payload {
  ...
} with { encode "PayloadCodec1"; encode "PayloadCodec2" }

// PDU type with two encoding options
type record PDU {
  charstring source,
  charstring destination,
  Payload payload
} with { encode "PduCodec1"; encode "PduCodec2" }

template PDU m_msg := {
  source := "source address",
  destination := "destination address",
  payload := { ... }
}

testcase TC01() runs on C {
  p.setencode(PDU.payload, "PayloadCodec2");
  p.setencode (PDU, "PduCodec1");
  p.send(m_msg); // m_msg will be sent with its encode attribute set to "PduCodec1"
                 // and its payload field will have its encode attribute set to "PayloadCodec2"
  p.setencode (PDU, -); // resets encoding of the PDU to the original state (two supported
                        // encodings), the payload field will remain set to "PayloadCodec2"
}
```

Annex A (normative): BNF and static semantics

A.1 TTCN-3 BNF

A.1.0 General

This annex defines the syntax of TTCN-3 using extended BNF (henceforth just called BNF).

A.1.1 Conventions for the syntax description

Table A.1 defines the metanotation used to specify the extended BNF grammar for TTCN-3.

Table A.1: The syntactic metanotation

::=	is defined to be	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
{abc}#(n, m)	n to m instances of abc	repetition 3
(...)	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 shall only contain lowercase letters (a-z) uppercase letters (A-Z), numeric digits (0-9) and the underscore (_) symbol. 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*> or <end-of-file>.

EXAMPLE 2:

```
// This is a line comment
// spread over two lines
```

EXAMPLE 3:

```
// The following is not legal
const // This is MyConst integer c_myConst := 1;
// A block comment should have been used instead
const /* This is MyConst */ integer c_myConst := 1;
// A line comment like this works as well
const // This is MyConst
    integer c_myConst := 1;
```

A.1.5 TTCN-3 terminals

A.1.5.0 General

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	**( )**
Element specifier symbols	**[ ]**
Range symbol	**..**
Line and block comments	**/* */ //**
Statement separator symbol	**;**
Arithmetic operator symbols	**+ / - ***
Concatenation operator symbol	**&**
Relational operator symbols	**!= == >= <= < >**
Shift operator symbols	**<< >>**
Rotate operator symbols	**<@ @>**
String enclosure symbols	**" '**
Wildcard/matching symbols	**? ***
Assignment symbol	**:=**
Communication operation assignment	**->**
Bitstring, hexstring and Octetstring values	**B H O**
Float exponent	**E**
List element separator symbol	**,**
Field reference	**.**
Decoded field reference	**=>**

The predefined function identifiers defined in table 14 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**		
decmatch	**length**	**recursive**	**while**
default	**log**	**rem**	**with**
disconnect		**repeat**	
display	**map**	**reply**	**xor**
do	**match**	**return**	**xor4b**
done	**message**	**running**	
	mixed	**runs**	
else	**mod**		
encode	**modifies**		
enumerated	**module**		
error	**modulepar**		
except	**mtc**		
exception			
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.

Additionally, there are special TTCN-3 terminals consisting of an @-symbol, directly followed by an identifier. These terminals shall also be written in all lowercase letters.

NOTE: These terminals can be used in combination with the @-symbol, which results in a specific semantics for the annotated language element. They can also be used like any other identifier without any special meaning.

Table A.4: List of TTCN-3 terminals which are modifiers

@decoded **@default** **@deterministic**	**@fuzzy** **@index**	**@lazy** **@local**	**@nocase**

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 Recommendation ITU-T T.50 [4] and of annex A of Recommendation ITU-T 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 Recommendation ITU-T T.50 [4] and of annex A of Recommendation ITU-T 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 StructDefBody
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 ::= [DefaultModifier] (Type | NestedTypeDef) Identifier [ArrayDef] [SubTypeSpec]
/** STATIC SEMANTICS: at most one UnionFieldDef of UnionDefBody or NestedUnionDef shall contain a
DefaultModifier */
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 ["(" IntegerValueOrRange {"," IntegerValueOrRange } ")"]
39.IntegerValueOrRange ::= IntegerValue [".." IntegerValue]
40.IntegerValue ::= [Minus] Number
41.SubTypeDef ::= Type (Identifier | AddressKeyword) [ArrayDef] [SubTypeSpec]
42.SubTypeSpec ::= AllowedValuesSpec [StringLength] | StringLength
```

```
/* STATIC SEMANTICS - AllowedValues shall be of the same type as the field being subtyped */
43.AllowedValuesSpec ::= "(" ((TemplateOrRange {"," TemplateOrRange}) |
                              CharStringMatch) ")"
44.TemplateOrRange ::= RangeDef |
                       TemplateBody |
                       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 */
45.RangeDef ::= Bound ".." Bound
46.StringLength ::= LengthKeyword "(" SingleExpression [".."(SingleExpression | InfinityKeyword) ]
")"

/* 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) */
47.LengthKeyword ::= "length"
48.PortDef ::= PortKeyword PortDefBody
49.PortDefBody ::= Identifier PortDefAttribs
50.PortKeyword ::= "port"
51.PortDefAttribs ::= MessageAttribs |
                      ProcedureAttribs |
                      MixedAttribs
52.MessageAttribs ::= MessageKeyword "{" {(AddressDecl |
                                           MessageList |
                                           ConfigParamDef
                                          ) [SemiColon]}+ "}"
53.ConfigParamDef ::= MapParamDef | UnmapParamDef
54.MapParamDef ::= MapKeyword ParamKeyword "(" FormalValuePar {"," FormalValuePar}
                   ")"
55.UnmapParamDef ::= UnmapKeyword ParamKeyword "(" FormalValuePar {","
                                                                   FormalValuePar}
                     ")"
56.AddressDecl ::= AddressKeyword Type
57.MessageList ::= Direction AllOrTypeList
58.Direction ::= InParKeyword |
                 OutParKeyword |
                 InOutParKeyword
59.MessageKeyword ::= "message"
60.AllOrTypeList ::= AllKeyword | TypeList

/* NOTE: The use of AllKeyword in port definitions is deprecated */
61.AllKeyword ::= "all"
62.TypeList ::= Type {"," Type}
63.ProcedureAttribs ::= ProcedureKeyword "{" {(AddressDecl |
                                               ProcedureList |
                                               ConfigParamDef
                                              ) [SemiColon]}+ "}"
64.ProcedureKeyword ::= "procedure"
65.ProcedureList ::= Direction AllOrSignatureList
66.AllOrSignatureList ::= AllKeyword | SignatureList
67.SignatureList ::= Signature {"," Signature}
68.MixedAttribs ::= MixedKeyword "{" {(AddressDecl |
                                      MixedList |
                                      ConfigParamDef
                                     ) [SemiColon]}+ "}"
69.MixedKeyword ::= "mixed"
70.MixedList ::= Direction ProcOrTypeList
71.ProcOrTypeList ::= AllKeyword | (ProcOrType {"," ProcOrType})
72.ProcOrType ::= Signature | Type
73.ComponentDef ::= ComponentKeyword Identifier [ExtendsKeyword ComponentType
                                                 {"," ComponentType}] "{"
                    [ComponentDefList] "}"
74.ComponentKeyword ::= "component"
75.ExtendsKeyword ::= "extends"
76.ComponentType ::= ExtendedIdentifier
77.ComponentDefList ::= {ComponentElementDef [WithStatement] [SemiColon]}
78.ComponentElementDef ::= PortInstance |
                           VarInstance |
                           TimerInstance |
                           ConstDef |
                           TemplateDef
79.PortInstance ::= PortKeyword ExtendedIdentifier PortElement {"," PortElement}
```

```
80.PortElement ::= Identifier [ArrayDef]
```

A.1.6.1.2 Constant definitions

```
81.ConstDef ::= ConstKeyword Type ConstList
82.ConstList ::= SingleConstDef {"," SingleConstDef}
83.SingleConstDef ::= Identifier [ArrayDef] AssignmentChar ConstantExpression
84.ConstKeyword ::= "const"
```

A.1.6.1.3 Template definitions

```
85.TemplateDef ::= TemplateKeyword [TemplateRestriction] [FuzzyModifier]
                   BaseTemplate [DerivedDef] AssignmentChar TemplateBody
86.BaseTemplate ::= (Type | Signature) Identifier ["(" TemplateOrValueFormalParList
                                                   ")"]
87.TemplateKeyword ::= "template"
88.DerivedDef ::= ModifiesKeyword ExtendedIdentifier
89.ModifiesKeyword ::= "modifies"
90.TemplateOrValueFormalParList ::= TemplateOrValueFormalPar {"," TemplateOrValueFormalPar}
91.TemplateOrValueFormalPar ::= FormalValuePar | FormalTemplatePar

/* STATIC SEMANTICS - FormalValuePar shall resolve to an in parameter */
92.TemplateBody ::= (SimpleSpec |
                     FieldSpecList |
                     ArrayValueOrAttrib
                    ) [ExtraMatchingAttributes]

/* STATIC SEMANTICS - Within TeplateBody the ArrayValueOrAttrib can be used for array, record,
record of and set of types. */
93.SimpleSpec ::= (SingleExpression ["&" SimpleTemplateSpec]) | SimpleTemplateSpec
94.SimpleTemplateSpec ::= SingleTemplateExpression ["&" SimpleSpec]
95.SingleTemplateExpression ::= MatchingSymbol |
                                {TemplateRefWithParList [ExtendedFieldReference]) |
                                 ExtendedIdentifier EnumTemplateExtension
/** STATIC Semantics: ExtendedIdentifier shall refer to an enumerated value with associated value */
96.EnumTemplateExtension ::= "(" TemplateBody {"," TemplateBody } ")"
/** STATIC Semantics: each TemplateBody shall be an integer template */
97.FieldSpecList ::= "{" FieldSpec {"," FieldSpec} "}"
98.FieldSpec ::= FieldReference AssignmentChar (TemplateBody | Minus)
99.FieldReference ::= StructFieldRef |
                      ArrayOrBitRef |
                      ParRef
100.StructFieldRef ::= Identifier |
                       PredefinedType |
                       TypeReference

/* STATIC SEMANTICS - PredefinedType and TypeReference shall be used for anytype value notation
only. PredefinedType shall not be AnyTypeKeyword.*/
101.ParRef ::= Identifier

/* STATIC SEMANTICS - Identifier in ParRef shall be a formal parameter identifier from the
associated signature definition */
102.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 */
103.FieldOrBitNumber ::= SingleExpression

/* STATIC SEMANTICS - SingleExpression will resolve to a value of integer type */
104.ArrayValueOrAttrib ::= "{" [ArrayElementSpecList] "}"
105.ArrayElementSpecList ::= ArrayElementSpec {"," ArrayElementSpec}
106.ArrayElementSpec ::= Minus |
                         PermutationMatch |
                         TemplateBody
107.MatchingSymbol ::= Complement |
                       (AnyValue [WildcardLengthMatch]) |
                       (AnyOrOmit [WildcardLengthMatch]) |
                       ListOfTemplates |
                       Range |
                       BitStringMatch |
                       HexStringMatch |
                       OctetStringMatch |
                       CharStringMatch |
                       SubsetMatch |
                       SupersetMatch |
```

```
                        DecodedContentMatch
108.DecodedContentMatch ::= DecodedMatchKeyword ["(" [Expression] ")"] TemplateInstance
109.DecodedMatchKeyword ::= "decmatch"


/* STATIC SEMANTIC – WildcardLengthMatch shall 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, ListOfTemplates, Range, BitStringMatch, HexStringMatch, OctetStringMatch,
CharStringMatch, SubsetMatch and SupersetMatch productions shall not be used. */
110.ExtraMatchingAttributes ::= StringLength |
                               IfPresentKeyword |
                               (StringLength IfPresentKeyword)
111.BitStringMatch ::= "'" {BinOrMatch} "'" "B"
112.BinOrMatch ::= Bin |
                   AnyValue |
                   AnyOrOmit
113.HexStringMatch ::= "'" {HexOrMatch} "'" "H"
114.HexOrMatch ::= Hex |
                   AnyValue |
                   AnyOrOmit
115.OctetStringMatch ::= "'" {OctOrMatch} "'" "O"
116.OctOrMatch ::= Oct |
                   AnyValue |
                   AnyOrOmit
117.CharStringMatch ::= PatternKeyword [CaseInsenModifier] PatternParticle {"&" PatternParticle}
118.PatternParticle ::= Pattern | ReferencedValue
119.PatternKeyword ::= "pattern"
120.Pattern ::= """ {PatternElement} """
121.PatternElement ::= (("\" ("?" | "*" | "\" | "[" | "]" | "{" | "}" |
                             """ | "|" | "(" | ")" | "#" | "+" | "d" |
                             "w" | "t" | "n" | "r" | "s" | "b"
                            )) | ("?" | "*" | "\" | "|" | "+"
                                 ) | ("[" ["^"] [{PatternClassChar ["-"
                                                                   PatternClassChar]}]
                                       "]") |
                        ("{" ["\"] ReferencedValue "}") | ("\" "N" "{"
                                                          (ReferencedValue |
                                                          Type) "}") |
                        (""" """) |
                        ("(" PatternElement ")") |
                        ("#" (Num |
                             ("(" Number "," [Number] ")") |
                             ("(" "," Number ")") |
                             ("(" ["," ] ")") Num ")"
                            ))
                       ) | PatternChar
122.PatternChar ::= NonSpecialPatternChar | PatternQuadruple

/* STATIC SEMANTICS: Characters "?", "*", "\", "[", "]", "{", "}", """, "|", "(", ")", "#", "+",
"d", "^", "N" have special semantics – they are metacharacters for the definition of pattern
elements – only if they follow the BNF as defined above, if not they are interpreted like normal
characters */
123.NonSpecialPatternChar ::= Char
124.PatternClassChar ::= NonSpecialPatternClassChar |
                         PatternQuadruple |
                         "\" EscapedPatternClassChar
125.NonSpecialPatternClassChar ::= Char

/* STATIC SEMANTICS: Characters "[", "-", "^", "]", "\", "q", ","have special semantics – they are
metacharacters for the definition of pattern class characters – only if they follow the BNF as
defined above, if not they are interpreted like normal characters */
126.EscapedPatternClassChar ::= "[" | "-" | "^" | "]"
127.PatternQuadruple ::= "\" "q" "(" Number "," Number "," Number ","
                         Number ")"
128.Complement ::= ComplementKeyword ListOfTemplates
129.ComplementKeyword ::= "complement"
130.ListOfTemplates ::= "(" TemplateListItem {"," TemplateListItem} ")"
131.TemplateListItem ::= TemplateBody | AllElementsFrom
132.AllElementsFrom ::= AllKeyword FromKeyword TemplateBody
133.SubsetMatch ::= SubsetKeyword ListOfTemplates
134.SubsetKeyword ::= "subset"
135.SupersetMatch ::= SupersetKeyword ListOfTemplates
136.SupersetKeyword ::= "superset"
137.PermutationMatch ::= PermutationKeyword ListOfTemplates

/* STATIC SEMANTICS: Restrictions on the content of TemplateBody within the ListOfTemplates are
given in clause B.1.3.3. */
138.PermutationKeyword ::= "permutation"
```

```
139.AnyValue ::= "?"
140.AnyOrOmit ::= "*"
141.WildcardLengthMatch ::= LengthKeyword "(" SingleExpression ")"

/* STATIC SEMANTICS: SingleExpression shall evaluate to type integer */
142.IfPresentKeyword ::= "ifpresent"
143.PresentKeyword ::= "present"
144.Range ::= "(" Bound ".." Bound ")"
145.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. */
146.InfinityKeyword ::= "infinity"
147.ActualParAssignment ::= Identifier ":=" TemplateInstance
/* STATIC SEMANTICS – if a value parameter is used, an in-line template shall evaluate to a value */
148.TemplateRefWithParList ::= ExtendedIdentifier [ActualParList]
149.TemplateInstance ::= [(Type | Signature) Colon] [DerivedRefWithParList AssignmentChar]
                        TemplateBody
150.DerivedRefWithParList ::= ModifiesKeyword TemplateRefWithParList
151.ActualParList ::= "(" [(ActualPar {"," ActualPar })
                                  {"," ActualParAssignment}) |
                             (ActualParAssignment {"," ActualParAssignment})]
                          ")"
152.ActualPar ::= TemplateInstance | Minus

/* STATIC SEMANTICS - When the corresponding formal parameter is not of template type the
TemplateInstance production shall resolve to one or more SingleExpressions */
153.TemplateOps ::= MatchOp | ValueofOp
154.MatchOp ::= MatchKeyword "(" Expression "," TemplateInstance ")"
155.MatchKeyword ::= "match"
156.ValueofOp ::= ValueofKeyword "(" TemplateInstance")"
157.ValueofKeyword ::= "valueof"
```

A.1.6.1.4 Function definitions

```
158.FunctionDef ::= FunctionKeyword [DeterministicModifier] Identifier
                    "(" [FunctionFormalParList] ")" [RunsOnSpec] [MtcSpec]
                    [SystemSpec] [ReturnType] StatementBlock
159.FunctionKeyword ::= "function"
160.FunctionFormalParList ::= FunctionFormalPar {"," FunctionFormalPar}
161.FunctionFormalPar ::= FormalValuePar |
                          FormalTimerPar |
                          FormalTemplatePar |
                          FormalPortPar
162.ReturnType ::= ReturnKeyword [TemplateKeyword | RestrictedTemplate]
                   Type
163.ReturnKeyword ::= "return"
164.RunsOnSpec ::= RunsKeyword OnKeyword ComponentType
165.RunsKeyword ::= "runs"
166.OnKeyword ::= "on"
167.MtcSpec ::= MTCKeyword ComponentType
168.MTCKeyword ::= "mtc"
169.StatementBlock ::= "{" [FunctionDefList] [FunctionStatementList] "}"
170.FunctionDefList ::= {(FunctionLocalDef | FunctionLocalInst) [WithStatement]
                        [SemiColon]}+
171.FunctionStatementList ::= {FunctionStatement [SemiColon]}+
172.FunctionLocalInst ::= VarInstance | TimerInstance
173.FunctionLocalDef ::= ConstDef | TemplateDef
174.FunctionStatement ::= ConfigurationStatements |
                          TimerStatements |
                          CommunicationStatements |
                          BasicStatements |
                          BehaviourStatements |
                          SetLocalVerdict |
                          SUTStatements |
                          TestcaseOperation
175.FunctionInstance ::= FunctionRef "(" [ActualParList] ")"
176.FunctionRef ::= [Identifier Dot] (Identifier | PreDefFunctionIdentifier)
177.PreDefFunctionIdentifier ::= Identifier [CaseInsenModifier]

/* STATIC SEMANTICS - The Identifier shall be one of the pre-definedpredefined TTCN-3 function
identifiers from Annex C of ES 201 873-1. CaseInsenModifier shall be present only if Identifier is
"regexp". */
/* STATIC SEMANTICS – if a value parameter is used, an in-line template shall evaluate to a value */
```

A.1.6.1.5 Signature definitions

```
178.SignatureDef ::= SignatureKeyword Identifier "(" [SignatureFormalParList]
                     ")" [ReturnType | NoBlockKeyword] [ExceptionSpec]
179.SignatureKeyword ::= "signature"
180.SignatureFormalParList ::= FormalValuePar {"," FormalValuePar}
181.ExceptionSpec ::= ExceptionKeyword "(" TypeList ")"
182.ExceptionKeyword ::= "exception"
183.Signature ::= ExtendedIdentifier
184.NoBlockKeyword ::= "noblock"
```

A.1.6.1.6 Testcase definitions

```
185.TestcaseDef ::= TestcaseKeyword Identifier "(" [TemplateOrValueFormalParList]
                    ")" ConfigSpec StatementBlock
186.TestcaseKeyword ::= "testcase"
187.ConfigSpec ::= RunsOnSpec [SystemSpec]
188.SystemSpec ::= SystemKeyword ComponentType
189.SystemKeyword ::= "system"
190.TestcaseInstance ::= ExecuteKeyword "(" ExtendedIdentifier "(" [ActualParList]
                         ")" ["," (Expression | Minus) ["," SingleExpression]]
                         ")"
191.ExecuteKeyword ::= "execute"
```

A.1.6.1.7 Altstep definitions

```
192.AltstepDef ::= AltstepKeyword Identifier "(" [FunctionFormalParList]
                   ")" [RunsOnSpec] [MtcSpec] [SystemSpec] "{" AltstepLocalDefList
                   AltGuardList "}"
193.AltstepKeyword ::= "altstep"
194.AltstepLocalDefList ::= {AltstepLocalDef [WithStatement] [SemiColon]}
195.AltstepLocalDef ::= VarInstance |
                        TimerInstance |
                        ConstDef |
                        TemplateDef
196.AltstepInstance ::= ExtendedIdentifier "(" [ActualParList]
                        ")"
```

A.1.6.1.8 Import definitions

```
197.ImportDef ::= ImportKeyword ImportFromSpec (AllWithExcepts | ("{" ImportSpec "}"))
198.ImportKeyword ::= "import"
199.AllWithExcepts ::= AllKeyword [ExceptsDef]
200.ExceptsDef ::= ExceptKeyword "{" ExceptSpec "}"
201.ExceptKeyword ::= "except"
202.ExceptSpec ::= {ExceptElement [SemiColon]}
203.ExceptElement ::= ExceptGroupSpec |
                      ExceptTypeDefSpec |
                      ExceptTemplateSpec |
                      ExceptConstSpec |
                      ExceptTestcaseSpec |
                      ExceptAltstepSpec |
                      ExceptFunctionSpec |
                      ExceptSignatureSpec |
                      ExceptModuleParSpec
204.ExceptGroupSpec ::= GroupKeyword (QualifiedIdentifierList | AllKeyword)
205.IdentifierListOrAll ::= IdentifierList | AllKeyword
206.ExceptTypeDefSpec ::= TypeDefKeyword IdentifierListOrAll
207.ExceptTemplateSpec ::= TemplateKeyword IdentifierListOrAll
208.ExceptConstSpec ::= ConstKeyword IdentifierListOrAll
209.ExceptTestcaseSpec ::= TestcaseKeyword IdentifierListOrAll
210.ExceptAltstepSpec ::= AltstepKeyword IdentifierListOrAll
211.ExceptFunctionSpec ::= FunctionKeyword IdentifierListOrAll
212.ExceptSignatureSpec ::= SignatureKeyword IdentifierListOrAll
213.ExceptModuleParSpec ::= ModuleParKeyword IdentifierListOrAll
214.ImportSpec ::= {ImportElement [SemiColon]}
215.ImportElement ::= ImportGroupSpec |
                     ImportTypeDefSpec |
                     ImportTemplateSpec |
                     ImportConstSpec |
                     ImportTestcaseSpec |
                     ImportAltstepSpec |
                     ImportFunctionSpec |
                     ImportSignatureSpec |
```

```
                    ImportModuleParSpec |
                    ImportImportSpec
216.ImportFromSpec ::= FromKeyword ModuleId [RecursiveKeyword]
217.RecursiveKeyword ::= "recursive"
218.ImportGroupSpec ::= GroupKeyword (GroupRefListWithExcept | AllGroupsWithExcept)
219.GroupRefListWithExcept ::= QualifiedIdentifierWithExcept {"," QualifiedIdentifierWithExcept}
220.AllGroupsWithExcept ::= AllKeyword [ExceptKeyword QualifiedIdentifierList]
221.QualifiedIdentifierWithExcept ::= QualifiedIdentifier [ExceptsDef]
222.IdentifierListOrAllWithExcept ::= IdentifierList | AllWithExcept
223.ImportTypeDefSpec ::= TypeDefKeyword IdentifierListOrAllWithExcept
224.AllWithExcept ::= AllKeyword [ExceptKeyword IdentifierList]
225.ImportTemplateSpec ::= TemplateKeyword IdentifierListOrAllWithExcept
226.ImportConstSpec ::= ConstKeyword IdentifierListOrAllWithExcept
227.ImportAltstepSpec ::= AltstepKeyword IdentifierListOrAllWithExcept
228.ImportTestcaseSpec ::= TestcaseKeyword IdentifierListOrAllWithExcept
229.ImportFunctionSpec ::= FunctionKeyword IdentifierListOrAllWithExcept
230.ImportSignatureSpec ::= SignatureKeyword IdentifierListOrAllWithExcept
231.ImportModuleParSpec ::= ModuleParKeyword IdentifierListOrAllWithExcept
232.ImportImportSpec ::= ImportKeyword AllKeyword
```

A.1.6.1.9 Group definitions

```
233.GroupDef ::= GroupKeyword Identifier "{" [ModuleDefinitionsList] "}"
234.GroupKeyword ::= "group"
```

A.1.6.1.10 External function definitions

```
235.ExtFunctionDef ::= ExtKeyword FunctionKeyword [DeterministicModifier]
                       Identifier "(" [FunctionFormalParList] ")" [ReturnType]
236.ExtKeyword ::= "external"
```

A.1.6.1.11 External constant definitions

```
237.ExtConstDef ::= ExtKeyword ConstKeyword Type IdentifierList
```

A.1.6.1.12 Module parameter definitions

```
238.ModuleParDef ::= ModuleParKeyword (ModulePar | ("{" MultitypedModuleParList
                                                    "}"))
239.ModuleParKeyword ::= "modulepar"
240.MultitypedModuleParList ::= {ModulePar [SemiColon]}
241.ModulePar ::= Type ModuleParList
242.ModuleParList ::= Identifier [AssignmentChar ConstantExpression] {","
                      Identifier [AssignmentChar ConstantExpression]}
```

A.1.6.1.13 Friend module definitions

```
243.FriendModuleDef ::= "friend" "module" IdentifierList [SemiColon]
```

A.1.6.2 Control part

```
244.ModuleControlPart ::= ControlKeyword "{" ModuleControlBody "}" [WithStatement]
                          [SemiColon]
245.ControlKeyword ::= "control"
246.ModuleControlBody ::= [ControlStatementOrDefList]
247.ControlStatementOrDefList ::= {ControlStatementOrDef [SemiColon]}+
248.ControlStatementOrDef ::= (FunctionLocalDef | FunctionLocalInst) [WithStatement] |
                              ControlStatement
249.ControlStatement ::=  TimerStatements |
                          BasicStatements |
                          BehaviourStatements |
                          SUTStatements |
                          StopKeyword
```

A.1.6.3 Local definitions

A.1.6.3.1 Variable instantiation

```
250.VarInstance ::= VarKeyword (([LazyModifier | FuzzyModifier] Type VarList) |
                                ((TemplateKeyword | RestrictedTemplate)
                                [LazyModifier | FuzzyModifier] Type TempVarList))
251.VarList ::= SingleVarInstance {"," SingleVarInstance}
252.SingleVarInstance ::= Identifier [ArrayDef] [AssignmentChar Expression]
253.VarKeyword ::= "var"
254.TempVarList ::= SingleTempVarInstance {"," SingleTempVarInstance}
255.SingleTempVarInstance ::= Identifier [ArrayDef] [AssignmentChar TemplateBody]
256.VariableRef ::= Identifier [ExtendedFieldReference]
```

A.1.6.3.2 Timer instantiation

```
257.TimerInstance ::= TimerKeyword VarList
258.TimerKeyword ::= "timer"
259.ArrayIdentifierRef ::= Identifier {ArrayOrBitRef}
```

A.1.6.4 Operations

A.1.6.4.1 Component operations

```
260.ConfigurationStatements ::= ConnectStatement |
                                MapStatement |
                                DisconnectStatement |
                                UnmapStatement |
                                DoneStatement |
                                KilledStatement |
                                StartTCStatement |
                                StopTCStatement |
                                KillTCStatement
261.ConfigurationOps ::= CreateOp |
                         SelfOp |
                         SystemKeyword |
                         MTCKeyword |
                         RunningOp |
                         AliveOp
262.CreateOp ::= ComponentType Dot CreateKeyword ["(" (SingleExpression |
                    Minus) ["," SingleExpression] ")"] [AliveKeyword]
263.SelfOp ::= "self"
264.DoneStatement ::= ComponentOrAny Dot DoneKeyword [ PortRedirectSymbol
                     [ ValueStoreSpec ] [ IndexSpec ] ]
/*STATIC SEMANTICS – If PortRedirectSymbol is present, at least one of ValueStoreSpec and IndexSpec
shall be present*/
265.ComponentOrAny ::= ComponentOrDefaultReference |
                       (AnyKeyword (ComponentKeyword | FromKeyword VariableRef)) |
                       (AllKeyword ComponentKeyword)
266.ValueStoreSpec ::= ValueKeyword VariableRef
267.IndexAssignment ::= PortRedirectSymbol IndexSpec
268.IndexSpec ::= IndexModifier ValueStoreSpec
269.KilledStatement ::= ComponentOrAny Dot KilledKeyword [ PortRedirectSymbol
                       [ ValueStoreSpec ] [ IndexSpec] ]
/*STATIC SEMANTICS – If PortRedirectSymbol is present, at least one of ValueStoreSpec and IndexSpec
shall be present*/
270.DoneKeyword ::= "done"
271.KilledKeyword ::= "killed"
272.RunningOp ::= ComponentOrAny Dot RunningKeyword [IndexAssignment]
273.RunningKeyword ::= "running"
274.AliveOp ::= ComponentOrAny Dot AliveKeyword [IndexAssignment]
275.CreateKeyword ::= "create"
276.AliveKeyword ::= "alive"
277.ConnectStatement ::= ConnectKeyword SingleConnectionSpec
278.ConnectKeyword ::= "connect"
279.SingleConnectionSpec ::= "(" PortRef "," PortRef ")"
280.PortRef ::= ComponentRef Colon ArrayIdentifierRef
281.ComponentRef ::= ComponentOrDefaultReference |
                    SystemKeyword |
                    SelfOp |
                    MTCKeyword
282.DisconnectStatement ::= DisconnectKeyword [SingleConnectionSpec |
                                               AllConnectionsSpec |
```

```
                                          AllPortsSpec |
                                          AllCompsAllPortsSpec
                                         ]
283.AllConnectionsSpec ::= "(" PortRef ")"
284.AllPortsSpec ::= "(" ComponentRef ":" AllKeyword PortKeyword ")"
285.AllCompsAllPortsSpec ::= "(" AllKeyword ComponentKeyword ":" AllKeyword
                             PortKeyword ")"
286.DisconnectKeyword ::= "disconnect"
287.MapStatement ::= MapKeyword SingleConnectionSpec [ParamClause]
288.ParamClause ::= ParamKeyword ActualParList
289.MapKeyword ::= "map"
290.UnmapStatement ::= UnmapKeyword [SingleConnectionSpec [ParamClause] |
                                    AllConnectionsSpec [ParamClause] |
                                    AllPortsSpec |
                                    AllCompsAllPortsSpec
                                   ]
291.UnmapKeyword ::= "unmap"
292.StartTCStatement ::= ComponentOrDefaultReference Dot StartKeyword
                         "(" (FunctionInstance | AltstepInstance)  ")"
293.StartKeyword ::= "start"
294.StopTCStatement ::= StopKeyword | (ComponentReferenceOrLiteral | AllKeyword
                                      ComponentKeyword) Dot StopKeyword
295.ComponentReferenceOrLiteral ::= ComponentOrDefaultReference |
                                    MTCKeyword |
                                    SelfOp
296.KillTCStatement ::= KillKeyword | ((ComponentReferenceOrLiteral |
                                        AllKeyword ComponentKeyword) Dot KillKeyword)
297.ComponentOrDefaultReference ::= VariableRef | FunctionInstance
298.KillKeyword ::= "kill"
```

A.1.6.4.2 Port operations

```
299.CommunicationStatements ::= SendStatement |
                                CallStatement |
                                ReplyStatement |
                                RaiseStatement |
                                ReceiveStatement |
                                TriggerStatement |
                                GetCallStatement |
                                GetReplyStatement |
                                CatchStatement |
                                CheckStatement |
                                ClearStatement |
                                StartStatement |
                                StopStatement |
                                HaltStatement |
                                CheckStateStatement
300.SendStatement ::= ArrayIdentifierRef Dot PortSendOp
301.PortSendOp ::= SendOpKeyword "(" TemplateInstance")" [ToClause]
302.SendOpKeyword ::= "send"
303.ToClause ::= ToKeyword (TemplateInstance|
                            AddressRefList |
                            AllKeyword ComponentKeyword
                           )
304.AddressRefList ::= "(" TemplateInstance{"," TemplateInstance} ")"
305.ToKeyword ::= "to"
306.CallStatement ::= ArrayIdentifierRef Dot PortCallOp [PortCallBody]
307.PortCallOp ::= CallOpKeyword "(" CallParameters ")" [ToClause]
308.CallOpKeyword ::= "call"
309.CallParameters ::= TemplateInstance ["," CallTimerValue]
310.CallTimerValue ::= Expression | NowaitKeyword
311.NowaitKeyword ::= "nowait"
312.PortCallBody ::= "{" CallBodyStatementList "}"
313.CallBodyStatementList ::= {CallBodyStatement [SemiColon]}+
314.CallBodyStatement ::= CallBodyGuard StatementBlock
315.CallBodyGuard ::= AltGuardChar CallBodyOps
316.CallBodyOps ::= GetReplyStatement | CatchStatement
317.ReplyStatement ::= ArrayIdentifierRef Dot PortReplyOp
318.PortReplyOp ::= ReplyKeyword "(" TemplateInstance [ReplyValue] ")" [ToClause]
319.ReplyKeyword ::= "reply"
320.ReplyValue ::= ValueKeyword TemplateBody
/* STATIC SEMANTICS - TemplateBody shall be type compatible with the return type. It shall evaluate
to a value or template (literal or template instance)  conforming to the template(value)
restriction. */
321.RaiseStatement ::= ArrayIdentifierRef Dot PortRaiseOp
322.PortRaiseOp ::= RaiseKeyword "(" Signature "," TemplateInstance")"
                   [ToClause]
```

```
323.RaiseKeyword ::= "raise"
324.ReceiveStatement ::= PortOrAny Dot PortReceiveOp
325.PortOrAny ::= ArrayIdentifierRef | (AnyKeyword (PortKeyword | FromKeyword VariableRef))
326.PortReceiveOp ::= ReceiveOpKeyword ["("TemplateInstance")"] [FromClause] [PortRedirect]
327.ReceiveOpKeyword ::= "receive"
328.FromClause ::= FromKeyword (TemplateInstance |
                               AddressRefList |
                               AnyKeyword ComponentKeyword
                              )
329.FromKeyword ::= "from"
330.PortRedirect ::= PortRedirectSymbol ((ValueSpec [SenderSpec] [IndexSpec]) |
                                        (SenderSpec [IndexSpec]) |
                                         IndexSpec
                                        )
331.PortRedirectSymbol ::= "->"
332.ValueSpec ::= ValueKeyword (VariableRef | ("(" SingleValueSpec {"," SingleValueSpec} ")"))
333.SingleValueSpec ::= VariableRef [AssignmentChar [ DecodedModifier ["(" [Expression] ")"] ]
                                                    FieldReference ExtendedFieldReference]

/*STATIC SEMANTICS – FieldReference shall not be ParRef and ExtendedFieldReference shall not be
TypeDefIdentifier*/
334.ValueKeyword ::= "value"
335.SenderSpec ::= SenderKeyword VariableRef
336.SenderKeyword ::= "sender"
337.TriggerStatement ::= PortOrAny Dot PortTriggerOp
338.PortTriggerOp ::= TriggerOpKeyword ["(" TemplateInstance  ")"] [FromClause]
                      [PortRedirect]
339.TriggerOpKeyword ::= "trigger"
340.GetCallStatement ::= PortOrAny Dot PortGetCallOp
341.PortGetCallOp ::= GetCallOpKeyword ["(" TemplateInstance ")"] [FromClause]
                      [PortRedirectWithParam]
342.GetCallOpKeyword ::= "getcall"
343.PortRedirectWithParam ::= PortRedirectSymbol RedirectWithParamSpec
344.RedirectWithParamSpec ::= (ParamSpec [SenderSpec] [IndexSpec]) |
                              (SenderSpec [IndexSpec]) |
                              IndexSpec
345.ParamSpec ::= ParamKeyword ParamAssignmentList
346.ParamKeyword ::= "param"
347.ParamAssignmentList ::= "(" (AssignmentList | VariableList) ")"
348.AssignmentList ::= VariableAssignment {"," VariableAssignment}
349.VariableAssignment ::= VariableRef AssignmentChar [ DecodedModifier ["(" Expression] ")"]
                                                      Identifier
350.VariableList ::= VariableEntry {"," VariableEntry}
351.VariableEntry ::= VariableRef | Minus
352.GetReplyStatement ::= PortOrAny Dot PortGetReplyOp
353.PortGetReplyOp ::= GetReplyOpKeyword ["(" TemplateInstance [ValueMatchSpec]
                                          ")"] [FromClause] [PortRedirectWithValueAndParam]
354.PortRedirectWithValueAndParam ::= PortRedirectSymbol RedirectWithValueAndParamSpec
355.RedirectWithValueAndParamSpec ::= (ValueSpec [ParamSpec] [SenderSpec]
                                      [IndexSpec]) | RedirectWithParamSpec
356.GetReplyOpKeyword ::= "getreply"
357.ValueMatchSpec ::= ValueKeyword TemplateInstance
358.CheckStatement ::= PortOrAny Dot PortCheckOp
359.PortCheckOp ::= CheckOpKeyword ["(" CheckParameter ")"]
```

```
360.CheckOpKeyword ::= "check"
361.CheckParameter ::= CheckPortOpsPresent |
                       FromClausePresent |
                       RedirectPresent
362.FromClausePresent ::= FromClause [PortRedirectSymbol ((SenderSpec
                                                           [IndexSpec]) |
                                                          IndexSpec)]
363.RedirectPresent ::= PortRedirectSymbol ((SenderSpec [IndexSpec]) |
                                            IndexSpec)
364.CheckPortOpsPresent ::= PortReceiveOp |
                            PortGetCallOp |
                            PortGetReplyOp |
                            PortCatchOp
365.CatchStatement ::= PortOrAny Dot PortCatchOp
366.PortCatchOp ::= CatchOpKeyword ["(" CatchOpParameter ")"] [FromClause] [PortRedirect]
367.CatchOpKeyword ::= "catch"
368.CatchOpParameter ::= Signature "," TemplateInstance | TimeoutKeyword
369.ClearStatement ::= PortOrAll Dot ClearOpKeyword
370.PortOrAll ::= ArrayIdentifierRef | AllKeyword PortKeyword
371.ClearOpKeyword ::= "clear"
372.StartStatement ::= PortOrAll Dot StartKeyword
373.StopStatement ::= PortOrAll Dot StopKeyword
374.StopKeyword ::= "stop"
375.HaltStatement ::= PortOrAll Dot HaltKeyword
376.HaltKeyword ::= "halt"
377.AnyKeyword ::= "any"
378.CheckStateStatement ::= PortOrAllAny Dot CheckStateKeyword "(" SingleExpression
                          ")"
379.PortOrAllAny ::= PortOrAll | AnyKeyword PortKeyword
380.CheckStateKeyword ::= "checkstate"
```

A.1.6.4.3 Timer operations

```
381.TimerStatements ::= StartTimerStatement |
                        StopTimerStatement |
                        TimeoutStatement
382.TimerOps ::= ReadTimerOp | RunningTimerOp
383.StartTimerStatement ::= ArrayIdentifierRef Dot StartKeyword ["(" Expression ")"]
384.StopTimerStatement ::= TimerRefOrAll Dot StopKeyword
385.TimerRefOrAll ::= ArrayIdentifierRef | AllKeyword TimerKeyword
386.ReadTimerOp ::= ArrayIdentifierRef Dot ReadKeyword
387.ReadKeyword ::= "read"
388.RunningTimerOp ::= TimerRefOrAny Dot RunningKeyword [IndexAssignment]
389.TimeoutStatement ::= TimerRefOrAny Dot TimeoutKeyword [IndexAssignment]
390.TimerRefOrAny ::= ArrayIdentifierRef |
                    (AnyKeyword TimerKeyword) |
                    (AnyKeyword FromKeyword Identifier)
391.TimeoutKeyword ::= "timeout"
```

A.1.6.4.4 Testcase operation

```
392.TestcaseOperation ::= TestcaseKeyword "." StopKeyword ["(" { LogItem [","] } ")"]
```

A.1.6.5 Type

```
393.Type ::= PredefinedType | ReferencedType
394.PredefinedType ::= BitStringKeyword |
                       BooleanKeyword |
                       CharStringKeyword |
                       UniversalCharString |
                       IntegerKeyword |
                       OctetStringKeyword |
                       HexStringKeyword |
                       VerdictTypeKeyword |
                       FloatKeyword |
                       AddressKeyword |
                       DefaultKeyword |
                       AnyTypeKeyword
395.BitStringKeyword ::= "bitstring"
396.BooleanKeyword ::= "boolean"
397.IntegerKeyword ::= "integer"
398.OctetStringKeyword ::= "octetstring"
399.HexStringKeyword ::= "hexstring"
400.VerdictTypeKeyword ::= "verdicttype"
401.FloatKeyword ::= "float"
```

```
402.AddressKeyword ::= "address"
403.DefaultKeyword ::= "default"
404.AnyTypeKeyword ::= "anytype"
405.CharStringKeyword ::= "charstring"
406.UniversalCharString ::= UniversalKeyword CharStringKeyword
407.UniversalKeyword ::= "universal"
408.ReferencedType ::= ExtendedIdentifier [ExtendedFieldReference]
409.TypeReference ::= ExtendedIdentifier
410.ArrayDef ::= {"[" SingleExpression [".." SingleExpression] "]"}+

/* STATIC SEMANTICS - ArrayBounds will resolve to a non negative value of integer type */
```

A.1.6.6 Value

```
411.Value ::= PredefinedValue | ReferencedValue
412.PredefinedValue ::= Bstring |
                        BooleanValue |
                        CharStringValue |
                        Number | /* IntegerValue */
                        Ostring |
                        Hstring |
                        VerdictTypeValue |
                        FloatValue |
                        AddressValue |
                        OmitKeyword
413.BooleanValue ::= "true" | "false"
414.VerdictTypeValue ::= "pass" |
                         "fail" |
                         "inconc" |
                         "none" |
                         "error"
415.CharStringValue ::= Cstring | Quadruple | USIlikeNotation
416.Quadruple ::= CharKeyword "(" Number "," Number "," Number "," Number ")"
417.USIlikeNotation ::= CharKeyword "(" UIDlike { "," UIDlike } ")"
418.UIDlike ::= ("U"|"u") {"+"} {Hex}#(1,8)
419.CharKeyword ::= "char"
420.FloatValue ::= FloatDotNotation |
                  FloatENotation |
                  NaNKeyword
421.NaNKeyword ::= "not_a_number"
422.FloatDotNotation ::= Number Dot DecimalNumber
423.FloatENotation ::= Number [Dot DecimalNumber] Exponential [Minus] Number
424.Exponential ::= "E"
425.ReferencedValue ::= ExtendedIdentifier [ExtendedFieldReference | ExtendedEnumReference]
/** STATIC Semantics: ExtendedEnumReference shall be present if and only if ExtendedIdentifier
refers to an enumerated value with an attached value list */
426.ExtendedEnumReference ::= "(" IntegerValue ")"
427.Number ::= (NonZeroNum {Num}) | "0"
428.NonZeroNum ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
429.DecimalNumber ::= { Num }+
430.Num ::= "0" | NonZeroNum
431.Bstring ::= "'" { Bin | BinSpace } "'" "B"
432.Bin ::= "0" | "1"
433.Hstring ::= "'" { Hex | BinSpace } "'" "H"
434.Hex ::= Num | "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" |
           "d" | "e" | "f"
435.Ostring ::= "'" { Oct | BinSpace } "'" "O"
436.Oct ::= Hex Hex
437.Cstring ::= """ {Char} """
438.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 */
439.Identifier ::= Alpha {AlphaNum | Underscore}
440.Alpha ::= UpperAlpha | LowerAlpha
441.AlphaNum ::= Alpha | Num
442.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"
443.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"
444.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)
*/
445.FreeText ::= """ {ExtendedAlphaNum} """
446.AddressValue ::= "null"
```

```
447.OmitKeyword ::= "omit"
448. BinSpace ::=  " " | "\" NLChar
449. NLChar ::= /* REFERENCE - Any sequence of newline characters that constitute a newline by using
the following C0 control characters: LF(10), VT(11), FF(12), CR(13) (see Recommendation ITU-T T.50
[4]) (jointly called newline characters, see clause A.1.5.1) from the character set defined in
Recommendation ITU-T T.50 [4].*/
```

A.1.6.7 Parameterization

```
450.InParKeyword ::= "in"
451.OutParKeyword ::= "out"
452.InOutParKeyword ::= "inout"
453.FormalValuePar ::= [(InParKeyword |
                        InOutParKeyword |
                        OutParKeyword
                       )] [LazyModifier | FuzzyModifier] Type Identifier
                      [":=" (Expression | Minus)]
454.FormalPortPar ::= [InOutParKeyword] Identifier Identifier

/* The first Identifier refers to the port type. The second Identifier refers to the port parameter
identifier */
455.FormalTimerPar ::= [InOutParKeyword] TimerKeyword Identifier
456.FormalTemplatePar ::= [(InParKeyword |
                           OutParKeyword |
                           InOutParKeyword
                          )] (TemplateKeyword | RestrictedTemplate) [LazyModifier |
                                                                    FuzzyModifier]
                         Type Identifier [":=" (TemplateInstance | Minus)]
457.RestrictedTemplate ::= OmitKeyword | (TemplateKeyword TemplateRestriction)
458.TemplateRestriction ::= "(" (OmitKeyword |
                                 ValueKeyword |
                                 PresentKeyword
                                ) ")"
```

A.1.6.8 Statements

A.1.6.8.1 With statement

```
459.WithStatement ::= WithKeyword WithAttribList
460.WithKeyword ::= "with"
461.WithAttribList ::= "{" MultiWithAttrib "}"
462.MultiWithAttrib ::= {SingleWithAttrib [SemiColon]}
463.SingleWithAttrib ::=    StandardAttribute |
                           VariantAttribute
464.StandardAttribute ::= AttribKeyword [OverrideKeyword | LocalModifier] [AttribQualifier]
                          FreeText
465. VariantAttribute ::= VariantKeyword [( OverrideKeyword | LocalModifier )]
                         [AttribQualifier] [ RelatedEncoding "." ] FreeText
466. RelatedEncoding ::= FreeText | ( "{" FreeText { "," FreeText } "}" )

467.AttribKeyword ::= EncodeKeyword |
                      DisplayKeyword |
                      ExtensionKeyword |
                      OptionalKeyword
468.EncodeKeyword ::= "encode"
469.VariantKeyword ::= "variant"
470.DisplayKeyword ::= "display"
471.ExtensionKeyword ::= "extension"
472.OverrideKeyword ::= "override"
473.LocalModifier ::= "@local"
474.AttribQualifier ::= "(" DefOrFieldRefList ")"
475.DefOrFieldRefList ::= DefOrFieldRef {"," DefOrFieldRef}
476.DefOrFieldRef ::= QualifiedIdentifier |
                      ((FieldReference | "[" Minus "]") [ExtendedFieldReference]) |
                      AllRef
477.QualifiedIdentifier ::= {Identifier Dot} Identifier
478.AllRef ::= (GroupKeyword AllKeyword [ExceptKeyword "{" QualifiedIdentifierList
                                        "}"]) | ((TypeDefKeyword |
                                                  TemplateKeyword |
                                                  ConstKeyword |
                                                  AltstepKeyword |
                                                  TestcaseKeyword |
                                                  FunctionKeyword |
                                                  SignatureKeyword |
```

```
                                                ModuleParKeyword
                                              ) AllKeyword [ExceptKeyword
                                                            "{" IdentifierList
                                                            "}"])
```

A.1.6.8.2 Behaviour statements

```
479.BehaviourStatements ::= TestcaseInstance |
                            FunctionInstance |
                            ReturnStatement |
                            AltConstruct |
                            InterleavedConstruct |
                            LabelStatement |
                            GotoStatement |
                            RepeatStatement |
                            DeactivateStatement |
                            AltstepInstance |
                            ActivateOp |
                            BreakStatement |
                            ContinueStatement
480.SetLocalVerdict ::= SetVerdictKeyword "(" SingleExpression {"," LogItem}
                         ")"
481.SetVerdictKeyword ::= "setverdict"
482.GetLocalVerdict ::= "getverdict"
483.SUTStatements ::= ActionKeyword "(" ActionText {StringOp ActionText}
                       ")"
484.ActionKeyword ::= "action"
485.ActionText ::= FreeText | Expression
486.ReturnStatement ::= ReturnKeyword [TemplateInstance]
/* STATIC SEMANTICS - TemplateInstance 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. */
487.AltConstruct ::= AltKeyword "{" AltGuardList "}"
488.AltKeyword ::= "alt"
489.AltGuardList ::= {GuardStatement | ElseStatement [SemiColon]}
490.GuardStatement ::= AltGuardChar (AltstepInstance [StatementBlock] |
                                   GuardOp StatementBlock)
491.ElseStatement ::= "[" ElseKeyword "]" StatementBlock
492.AltGuardChar ::= "[" [BooleanExpression] "]"
493.GuardOp ::= TimeoutStatement |
              ReceiveStatement |
              TriggerStatement |
              GetCallStatement |
              CatchStatement |
              CheckStatement |
              GetReplyStatement |
              DoneStatement |
              KilledStatement
494.InterleavedConstruct ::= InterleavedKeyword "{" InterleavedGuardList
                             "}"
495.InterleavedKeyword ::= "interleave"
496.InterleavedGuardList ::= {InterleavedGuardElement [SemiColon]}+
497.InterleavedGuardElement ::= InterleavedGuard StatementBlock
498.InterleavedGuard ::= "[" "]" GuardOp
499.LabelStatement ::= LabelKeyword Identifier
500.LabelKeyword ::= "label"
501.GotoStatement ::= GotoKeyword Identifier
502.GotoKeyword ::= "goto"
503.RepeatStatement ::= "repeat"
504.ActivateOp ::= ActivateKeyword "(" AltstepInstance ")"
505.ActivateKeyword ::= "activate"
506.DeactivateStatement ::= DeactivateKeyword ["(" ComponentOrDefaultReference
                                               ")"]
507.DeactivateKeyword ::= "deactivate"
508.BreakStatement ::= "break"
509.ContinueStatement ::= "continue"
```

A.1.6.8.3 Basic statements

```
510.BasicStatements ::= Assignment |
                        LogStatement |
                        LoopConstruct |
                        ConditionalConstruct |
                        SelectCaseConstruct |
                        StatementBlock
```

```
511.Expression ::= SingleExpression | CompoundExpression
512.CompoundExpression ::= FieldExpressionList | ArrayExpression

/* STATIC SEMANTICS - Within CompoundExpression the ArrayExpression can be used for Arrays, record,
record of and set of types. */
513.FieldExpressionList ::= "{" FieldExpressionSpec {"," FieldExpressionSpec}
                           "}"
514.FieldExpressionSpec ::= FieldReference AssignmentChar NotUsedOrExpression
515.ArrayExpression ::= "{" [ArrayElementExpressionList] "}"
516.ArrayElementExpressionList ::= NotUsedOrExpression {"," NotUsedOrExpression}
517.NotUsedOrExpression ::= Expression | Minus
518.ConstantExpression ::= SingleExpression | CompoundConstExpression
519.BooleanExpression ::= SingleExpression

/* STATIC SEMANTICS - BooleanExpression shall resolve to a Value of type Boolean */
520.CompoundConstExpression ::= FieldConstExpressionList | ArrayConstExpression

/* STATIC SEMANTICS - Within CompoundConstExpression the ArrayConstExpression can be used for
arrays, record, record of and set of types. */
521.FieldConstExpressionList ::= "{" FieldConstExpressionSpec {"," FieldConstExpressionSpec} "}"
522.FieldConstExpressionSpec ::= FieldReference AssignmentChar ConstantExpression
523.ArrayConstExpression ::= "{" [ArrayElementConstExpressionList] "}"
524.ArrayElementConstExpressionList ::= ConstantExpression {"," ConstantExpression}
525.Assignment ::= VariableRef AssignmentChar  TemplateBody
/* STATIC SEMANTICS - The Templatebody 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. */
526.SingleExpression ::= XorExpression {"or" XorExpression}

/* STATIC SEMANTICS - If more than one XorExpression exists, then the XorExpressions shall evaluate
to specific values of compatible types */
527.XorExpression ::= AndExpression {"xor" AndExpression}

/* STATIC SEMANTICS - If more than one AndExpression exists, then the AndExpressions shall evaluate
to specific values of compatible types */
528.AndExpression ::= NotExpression {"and" NotExpression}

/* STATIC SEMANTICS - If more than one NotExpression exists, then the NotExpressions shall evaluate
to specific values of compatible types */
529.NotExpression ::= ["not"] EqualExpression

/* STATIC SEMANTICS - Operands of the not operator shall be of type boolean or derivatives of type
Boolean. */
530.EqualExpression ::= RelExpression {EqualOp RelExpression}

/* STATIC SEMANTICS - If more than one RelExpression exists, then the RelExpressions shall evaluate
to specific values of compatible types. If only one RelExpression exists, it shall not derive to a
CompoundExpression. */
531.RelExpression ::= ShiftExpression [RelOp ShiftExpression] | CompoundExpression

/* STATIC SEMANTICS - If both ShiftExpressions exist, then each ShiftExpression shall evaluate to a
specific integer, Enumerated or float Value or derivatives of these types */
532.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 " */
533.BitOrExpression ::= BitXorExpression {"or4b" BitXorExpression}

/* STATIC SEMANTICS - If more than one BitXorExpression exists, then the BitXorExpressions shall
evaluate to specific values of compatible types */
534.BitXorExpression ::= BitAndExpression {"xor4b" BitAndExpression}

/* STATIC SEMANTICS - If more than one BitAndExpression exists, then the BitAndExpressions shall
evaluate to specific values of compatible types */
535.BitAndExpression ::= BitNotExpression {"and4b" BitNotExpression}

/* STATIC SEMANTICS - If more than one BitNotExpression exists, then the BitNotExpressions shall
evaluate to specific values of compatible types */
536.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. */
```

```
537.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. If only one MulExpression exists, it shall not derive to a CompoundExpression. */
538.MulExpression ::= UnaryExpression {MultiplyOp UnaryExpression} | CompoundExpression

/* 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. */
539.UnaryExpression ::= [UnaryOp] Primary

/* STATIC SEMANTICS - Primary shall resolve to a specific Value of type integer or float or
derivatives of these types.*/
540.Primary ::= OpCall |
                Value |
                "(" SingleExpression ")"
541.ExtendedFieldReference ::= {(Dot (Identifier | PredefinedType)) |
                                ArrayOrBitRef |
                                ("[" Minus "]") |
                                DecodedFieldReference
                               }+

/* 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. DecodedFieldReference shall not appear on the
LHS of assignments and in type references*/
542. DecodedFieldReference ::= "=>" DecodedFieldType
543. DecodedFieldType ::= PredefinedType |
                          Identifier |
                          "(" Type [ "," Expression ] ")"

/* The Identifier shall resolve into a type */

544.OpCall ::= ConfigurationOps |
               GetLocalVerdict |
               TimerOps |
               TestcaseInstance |
               (FunctionInstance [ExtendedFieldReference]) |
               (TemplateOps [ExtendedFieldReference]) |
               ActivateOp |
               GetAttributeOp

545.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) */
546.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) */
547.UnaryOp ::= "+" | "-"

/* STATIC SEMANTICS - Operands of the "+" or "-" operators shall be of type integer or float or
derivations of integer or float (i.e. subrange) */
548.RelOp ::= "<" | ">" | ">=" | "<="

/* STATIC SEMANTICS - the precedence of the operators is defined in Table 6 */
549.EqualOp ::= "==" | "!="
550.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 */
551.ShiftOp ::= "<<" | ">>" | "<@" | "@>"
552.LogStatement ::= LogKeyword "(" LogItem {"," LogItem} ")"
553.LogKeyword ::= "log"
554.LogItem ::= FreeText | TemplateInstance
555.LoopConstruct ::= ForStatement |
                     WhileStatement |
                     DoWhileStatement
556.ForStatement ::= ForKeyword "(" Initial SemiColon BooleanExpression
                    SemiColon Assignment ")" StatementBlock
557.ForKeyword ::= "for"
```

```
558.Initial ::= VarInstance | Assignment
559.WhileStatement ::= WhileKeyword "(" BooleanExpression ")" StatementBlock
560.WhileKeyword ::= "while"
561.DoWhileStatement ::= DoKeyword StatementBlock WhileKeyword "(" BooleanExpression
                         ")"
562.DoKeyword ::= "do"
563.ConditionalConstruct ::= IfKeyword "(" BooleanExpression ")" StatementBlock
                            {ElseIfClause} [ElseClause]
564.IfKeyword ::= "if"
565.ElseIfClause ::= ElseKeyword IfKeyword "(" BooleanExpression ")" StatementBlock
566.ElseKeyword ::= "else"
567.ElseClause ::= ElseKeyword StatementBlock
568.SelectCaseConstruct ::= SelectKeyword [UnionKeyword] "(" SingleExpression ")" SelectCaseBody
569.SelectKeyword ::= "select"
570.SelectCaseBody ::= "{" {SelectCase}+ [CaseElse] "}"
571.SelectCase ::= CaseKeyword ("("TemplateInstance {"," TemplateInstance}
                               ")" | ElseKeyword) StatementBlock
/** STATIC SEMANTICS TemplateInstance-s shall be Identifier-s if the UnionKeyword is present in the
surrounding SelectCaseConstruct (see clause 19.3.2)*/
572.CaseElse ::= CaseKeyword ElseKeyword StatementBlock
573.CaseKeyword ::= "case"
574.ExtendedIdentifier ::= [Identifier Dot] Identifier
/** STATIC SEMANTICS The optional Identifier Dot part shall not be used for enumerated values*/
575.IdentifierList ::= Identifier {"," Identifier}
576.QualifiedIdentifierList ::= QualifiedIdentifier {"," QualifiedIdentifier}
577.GetAttributeOp ::= (Type | TemplateInstance) "." GetAttributeSpec
578.GetAttributeSpec ::= EncodeKeyword |
                        VariantKeyword ["(" FreeText ")"] |
                        DisplayKeyword |
                        ExtensionKeyword |
                        OptionalKeyword
```

A.1.6.9 Miscellaneous productions

```
579.Dot ::= "."
580.Minus ::= "-"
581.SemiColon ::= ";"
582.Colon ::= ":"
583.Underscore ::= "_"
584.AssignmentChar ::= ":="
585.IndexModifier ::= "@index"
586.DeterministicModifier ::= "@deterministic"
587.LazyModifier ::= "@lazy"
588.FuzzyModifier ::= "@fuzzy"
589.CaseInsenModifier ::= "@nocase"
590.DecodedModifier ::= "@decoded"
591.DefaultModifier ::= "@default"
```

Annex B (normative): Matching values

B.1 Template matching mechanisms

B.1.0 General

This annex specifies the matching mechanisms that may be used in TTCN-3 templates (and only in templates).

B.1.1 Matching specific values

Specific values are the basic matching mechanism of TTCN-3 templates. Specific values in templates are expressions which do not contain any matching mechanisms or wildcards.

Unless otherwise specified, a template field matches the corresponding 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.

Restrictions

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

Examples

```
// 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 m_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

B.1.2.0 General

The following matching mechanisms may be used in place of explicit values.

B.1.2.1 Template list

A template list denotes a list of acceptable values. It can be used for values of all types. A template list may contain values, templates obeying the template (present) restriction (see clause 15.8), and members added by **`all from`** clauses. An **`all from`** clause comprises all elements of an existing **`record of`** or **`set of`** template into the template list.

A template field that uses a template list matches the corresponding field if, and only if, the field value matches any one of the values or templates in the template list, after resolving all **`all from`** clauses. Each value or template in the template list shall be of the type declared for the template field in which this mechanism is used.

Restrictions

a) The type of the template list and the member type of the template in the **`all from`** clause shall be compatible.

b) The template in the **`all from`** clause as a whole shall not resolve into a matching mechanism. Its elements may contain any of the matching mechanisms or matching attributes with the exception of those described in the following restriction.

c) Individual members of the template in the **`all from`** clause shall not resolve to any of the following matching mechanisms: *AnyElementsOrNone*, permutation.

d) Each value or template in the template list shall be of the type declared for the template field in which this mechanism is used.

e) Templates in the template list shall obey the template (present) restriction (see clause 15.8).

Examples

EXAMPLE 1:

```
template MyMessageType mw_myTemplate:=
{
    field1 := (2,4,6),                  // list of integer values
    field2 := ("String1", "String2"),   // list of charstring values
    :
    :
}
```

EXAMPLE 2:

```
type record of integer RoI;
template RoI mw_roI1 := {1, 2, (6..9)};

template RoI mw_roI2 := {1, *, 3};

template integer mw_i1 := (all from mw_roI1, 100);
// results in (1, 2, (6..9), 100)

template integer mw_i2 := (0, all from mw_roI2);
// causes an error because mw_roI2 contains AnyElementsOrNone

template RoI mw_roI3 := (all from mw_roI1);
// causes an error because member type of mw_roI1 (integer)
// is not compatible with the list template type (RoI)

template RoI mw_roI4 := ?;

template RoI mw_roI5 := (all from mw_roI4);
// causes an error, because mw_roI4 as a whole resolves into a matching mechanism
```

B.1.2.2 Complemented template list

The keyword **`complement`** denotes a list of values that will not be accepted as values (i.e. it is the complement of a template list). It can be used on all values of all types. A complemented value list may also contain templates obeying the present template restriction (see clause 15.8).

A template field that uses complement matches the corresponding field if and only if the corresponding field's value does not match any of the values or templates listed in the template list. The template list may be a single value, of course.

Besides specifying individual values, it is possible to add all elements of an existing **`record of`** or **`set of`** template into a complement template list using an **`all from`** clause.

Restrictions

a) The type of the complemented template list and the member type of the template in the **`all from`** clause shall be compatible.

b) The template in the **all from** clause as a whole shall not resolve into a matching mechanism (i.e. its elements may contain any of the matching mechanisms or matching attributes with the exception of those described in the following restriction).

c) Individual fields of the template in the **all from** clause shall not resolve to any of the following matching mechanisms: *AnyElementsOrNone*, permutation.

d) Each value or template in the list shall be of the type declared for the template field in which the complement is used.

e) The complement of a template list shall not match **omit**.

f) Templates in the complement of a template list shall obey the present template restriction (see clause 15.8).

Examples

EXAMPLE 1:

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

template MyMessageType mw_myTemplate:=
{
    field1 := complement (1,3,5),   // list of unacceptable integer values
    field2 := complement(true)              // will match false
}
```

EXAMPLE 2:

```
type record of integer RoI;

template RoI mw_roI1 := {1, 2, (6..9)};

template RoI mw_roI2 := {1, *, 3};

template integer mw_i1 := complement(all from mw_roI1, 100);
// matches integer values different from 1, 2, 6, 7, 8, 9 and 100

template integer mw_i2 := complement(0, all from mw_roI2);
// causes an error because mw_roI2 contains AnyElementsOrNone

template RoI mw_roI3 := complement(all from mw_roI1);
// causes an error because member type of mw_roI1 (integer) is not compatible
// with the complemented list template type (RoI)

template RoI mw_roI4 := ?;

template RoI mw_roI5 := complement (all from mw_roI4);
// causes an error because mw_roI4 resolves into a matching mechanism
```

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.

Restrictions

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

Examples

```
template MyMessageType mw_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 and the omission of the given optional field are acceptable. It can be assigned to templates of any type as a whole or to optional fields of **`set`** or **`record`** templates.

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.

Restrictions

a) It can be assigned to templates of any type as a whole or to optional fields of **`set`** or **`record`** templates.

b) At the time of matching during a receiving operation, it shall be applied to optional fields of record and set templates only.

Examples

```
type record MyMessageType2
{
  integer           field1,
  MyRecordofType        field2  optional,
  boolean           field3  optional
}

type record of integer MyRecordofType;

const MyMessageType2 c_myMessage := { {42}, omit, false }

template MyMessageType2 mw_myMessageTemplate:=
{
  :
  field3 := *       // matches true or false or omitted field3
}

template MyMessageType2 mw_myMessageTemplate2:=
{
  field1 := *,      // causes an error as field1 is mandatory
  :
}

template MyRecordofType mw_myRecofTemplate := *;    // this assignment is allowed
template boolean mw_myBoolTemplate := *;            // this assignment is allowed as well

template MyMessageType2 mw_myMessageTemplate3:=
{
  field1 := 42,
  field2 := mw_myRecofTemplate,
    // matches any valid value allowed by mw_myRecordof or absent field2
  field3 := mw_myBoolTemplate
    // matches true or false or omitted field3
}

v_mybooleanVar := match (c_myMessage.field2, mw_myRecofTemplate)
  // matches and returns true

v_mybooleanVar := match ({},mw_myRecofTemplate);
  // matches and returns true

v_mybooleanVar := match (false, mw_myBoolTemplate);
  // matches and returns true

v_mybooleanVar := match ({42,omit,true},mw_myMessageTemplate3);
  // matches and returns true
```

B.1.2.5 Value range

Ranges indicate a bounded range of acceptable values, including or excluding the boundaries.

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.

Restrictions

a) When used for values of **integer** or **float** types (and integer or float subtypes), a boundary value shall be either:

   1) infinity or -infinity;

   2) an expression that evaluates to a specific integer or float value.

b) 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.

c) 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).

d) Empty positions between the lower and the upper boundaries are not considered to be valid values of the specified range.

Examples

```
template MyMessageType mw_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 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 successful match shall be produced only if there exists such a one-to-one mapping from the *SuperSet* elements to the elements of the set of values where each *SuperSet* element matches the element of set of values it is mapped to. The *SuperSet* matching mechanism may contain templates (including template variables) obeying the present template restriction (see clause 15.8) and matching mechanisms with the restrictions given below. However, the length matching attribute may be attached to the *SuperSet* itself.

NOTE: The *SuperSet* matching mechanism imposes an implicit length restriction on the matched set of values: the set of values shall contain at least as many elements as the *SuperSet* template in order to produce a successful match.

Besides specifying individual values, it is possible to add all elements of a **record of** or **set of** template into *SuperSet*s using an **all from** clause.

Restrictions

a) *SuperSet* is an operation for matching that shall be used only on values of **set of** types.

b) Individual members of the *SuperSet's* argument shall be of the type replicated by the **set of**.

c) The member type of the set of associated with the *SuperSet* template and the member type of the template in the **all from** clause shall be compatible.

d) The template in the **all from** clause as a whole shall not resolve into a matching mechanism (i.e. its elements may contain any of the matching mechanisms or matching attributes with the exception of those described in the following restriction).

e) The individual members of the *SuperSet's* argument and the elements of the template in the **all from** clause shall not be the matching mechanisms omit, *SuperSet*, *SubSet* and the matching attributes (length restriction and ifpresent). In addition, the individual members shall not resolve to *AnyValueOrNone* and individual elements of the template in the **all from** clause shall not resolve to *AnyElementsOrNone* or permutation.

f) If the length matching attribute is attached to the *SuperSet*, the minimal length allowed by the length attribute shall not be less than the number of the elements in the *SuperSet*.

g) Templates in *SuperSet's* argument shall obey the present template restriction (see clause115.8).

Examples

EXAMPLE 1:

```
type set of integer MySetOfType (0 .. 10);

template MySetOfType mw_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 mw_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 mw_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 mw_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 mw_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 mw_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 at least 5 more valid integer values (i.e. between 0 and 10, inclusively) in
any order and position

template MySetOfType mw_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
```

EXAMPLE 2:

```
type record of integer RoI;
type set of integer SoI;
template RoI mw_roI1 := {1, 2, ?};

template SoI mw_soI1 := superset(all from mw_roI1);
// results in superset(1, 2, ?)
```

B.1.2.7 SubSet

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 successful match shall be produced only if there exists such a one-to-one mapping from the elements of the set of values to the *SubSet* elements where each element of the set of values is matched by the *SubSet* element it is mapped to. The *SubSet* matching mechanism may contain templates (including template variables) obeying the present template restriction (see clause 15.8) and matching mechanisms with the restrictions given below. However, the length matching attribute may be attached to the *SubSet* itself.

NOTE: The *SubSet* matching mechanism imposes an implicit length restriction on the matched set of values: the set of values shall contain at most as many elements as the *SubSet* template in order to produce a successful match.

Besides specifying individual values, it is possible to add all elements of a **record of** or **set of** template into *SubSet*s using an **all from** clause.

Restrictions

a) *SubSet* is an operation for matching that can be used only on values of **set of** types.

b) Individual members of the *SubSet's* argument shall be of the type replicated by the **set of**.

c) The member type of the set of type associated with the *SubSet* and the member type of the template in the **all from** clause shall be compatible.

d) The template in the **all from** clause as a whole shall not resolve into a matching mechanism (i.e. its elements may contain any of the matching mechanisms or matching attributes with the exception of those described in the following restriction).

e) The individual members of the *SubSet's* argument and the elements of the template in the **all from** clause shall not be the matching mechanisms omit, *SuperSet*, *SubSet* and the matching attributes (length restriction and ifpresent). In addition, individual members shall not resolve to *AnyValueOrNone* and individual fields of the template in the **all from** clause shall not resolve to *AnyElementsOrNone* or permutation.

f) If the length matching attribute is attached to the *SubSet*, the maximum length allowed by the length attribute shall not exceed the number of the elements in the *SubSet*.

g) Templates in *SubSet's* argument shall obey the present template restriction (see clause 15.8).

Examples

EXAMPLE 1:

```
template MySetOfType mw_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 mw_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 mw_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 mw_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 mw_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 mw_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 mw_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
```

EXAMPLE 2:

```
type record of integer RoI;
type set of integer SoI;
template RoI mw_roI1 := {1, 2, ?};

template SoI mw_soI1 := subset(all from mw_roI1);
// results in subset(1, 2, ?)
```

B.1.2.8 Omitting optional fields

The keyword **omit** denotes that an optional field shall be absent. If used as a matching mechanism, it matches an optional field if and only if it is absent.

Restrictions

a) It can be assigned to templates of any type as a whole or to optional fields of **set** or **record** templates.

b) At the time of matching during a receiving operation, it shall be applied to optional fields of **record** and **set** templates only.

Examples

```
type record MyMessageType2
{
  integer           field1,
  MyRecordofType        field2  optional,
  boolean           field3  optional
}

const MyMessageType2 c_myMessage := { 42, omit, false }

template MyMessageType2 m_myMessageTemplate:=
{
  :
  field3 := omit        // omits the optional field field3
}

template MyMessageType2 m_myMessageTemplate2:=
{
  field1 := omit,       // causes an error as field1 is mandatory
  :
}

template MyRecordof m_myRecofTemplate := omit;  // this assignment is allowed

template boolean m_myBoolTemplate := omit;      // this assignment is allowed as well

template MyMessageType2 m_myMessageTemplate3:=
{
  field1 := 42,
  field2 := m_myRecofTemplate,
    // matches if field2 is absent
  field3 := m_myBoolTemplate
    // matches if field3 is absent
}

v_myBooleanVar := match (c_myMessage.field2, m_myRecofTemplate)
  // matches and returns true

v_myBooleanVar := match ({},m_myRecofTemplate)
  // does not match and returns false

v_myBooleanVar := match (false, m_myBoolTemplate);
  // does not match and returns false

v_myBooleanVar := match ({42,omit,omit},m_myMessageTemplate3)
  // matches and returns true
```

B.1.2.9 Matching decoded content

The matching symbol *MatchDecodedContent* **decmatch** is used for checking encoded payload fields. The matching symbol is composed of the **decmatch** keyword, an optional encoding format parameter and a mandatory template instance called decoding target.

A template field that uses this symbol matches the corresponding field if, and only if, the field can be successfully decoded as an instance of the same type as the decoding target and if the decoded instance can be successfully matched by the decoding target.

The optional encoding format parameter may specify one of the UCS encoding formats (see clause C.5.4) that shall be used for the decoding trial, i.e. it overrides any variant attribute attached to the decoding target or the type of the decoding target (for example, for predefined variant attributes see clause 27.5).

Restrictions

a) It can be assigned to templates and template fields of **bitstring**, **hexstring**, **octetstring**, **charstring** and **universal charstring** types.

b) The decoding target can be a template of any data type.

c) The optional encoding format parameter can be used only for fields of r **universal charstring** types. The parameter value shall be of the **charstring** type and it shall contain one of the strings allowed for the **decvalue_unichar** predefined function (specified in clause C.5.4). Any other value shall cause an error.

d) If the template field is of **charstring** type or is of **universal charstring** type and the encoding format is missing, the default value "UTF-8" shall be used.

NOTE: The model of the behaviour of this implicit decoding is the following. At first, **hexstring** and **octetstring** values are implicitly converted to a **bitstring** value using the predefined **hex2bit** and **oct2bit** functions (specified in clauses C.1.18 and C.1.22) and **charstring** values are implicitly converted to **universal charstring** values. Prior to decoding, the **bitstring** and **universal charstring** values are stored into a temporary anonymous variable. Decoding is then performed by implicitly calling the predefined **decvalue** function (specified in clause C.5.2) for **bitstring** values and **decvalue_unichar** function for **universal charstring** values. The anonymous variable containing the encoded value is passed as the first parameter to the function, the second parameter contains another temporary variable called decoded instance. The decoded instance is of the same type as the decoding target. If the optional encoding format parameter is present, it is passed as the third parameter to the **decvalue_unichar** function. Decoding is successful only if the decoding function returns 0 and the first parameter contains an empty string (i.e. the whole encoded value has been successfully decoded). The matching mechanism will generate an unsuccessful match if decoding hasn't succeeded.

Examples

```
type record MyBinaryMessageType
{
  ...,
  octetstring   payload
}

type record MyTextMessageType
{
  ...,
  universal charstring  payload
}

type record MyPayloadType
{
  integer       field1,
  integer       field2
}

template MyBinaryMessageType mw_t1 :=
{
  :
  // The payload field can be matched only if it contains an encoded value of the MyPayload
  // type and if the field1 of the decoded value is equal to 1.
  payload := decmatch MyPayload:{field1 := 1, field2 := ? }
}

template MyTextMessageType mw_t2 :=
{
  :
  // The payload field can be matched only if it contains an encoded value of the MyPayloadType
  // type in the UTF-8 format and if the field1 of the decoded value is equal to 2 or 3.
  payload := decmatch("UTF-8") MyPayloadType:{field1 := (2, 3), field2 := ? }
}
```

B.1.2.10 Matching enumerated value with value list

To match an enumerated value with an associated value list in its definition, the enumerated value name shall be referenced followed by a non-empty list of integer templates in parenthesis.

The template matches only those enumerated values of the same name where the associated integer values is matched by at least one of the integer templates.

Examples

```
type enumerated Days
{
  Christmas(0), Easter(1), Other(2..365)
}

template integer mw_greater20 := complement(0 .. 20);
template mw_days1 := Other(5..6, greater20); // matches Other(5), Other(6) and
                                             // Other(21) .. Other(365)
template mw_days2 := Other(?); // matches Other(2) .. Other(365)
```

B.1.3 Matching mechanisms inside values

B.1.3.0 General

The following matching mechanisms may be used inside explicit values of strings, records, records of, sets, sets of and arrays.

B.1.3.1 Any element

B.1.3.1.0 General

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.

Restrictions

a) It shall be used only within values of string types, **`record of`** types, **`set of`** types and arrays.

Examples

```
template MyMessageType mw_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

B.1.3.2.0 General

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 "*".

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.

Restrictions

a) It shall be used only within values of string types, **record of** types, **set of** types and arrays and inside the permutation matching mechanism.

Examples

```
template MyMessageType mw_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
}

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

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** and array types. *Permutation* is denoted by the keyword **permutation**. *Permutation* elements shall obey the restrictions given below.

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).

Besides specifying all individual values, it is possible to add all elements of a **record of** or **set of** template into permutations using an **all from** clause.

Restrictions

a) Each individual member listed in the permutation shall be of the type replicated by the **record of** or array type.

b) The member type of the permutation and the member type of the template in the **all from** clause shall be compatible.

c) The template referenced in the **all from** clause as a whole shall not resolve into a matching mechanism other than a *SpecificValue* (see clause B.1.1), and it shall not contain permutations.

d) Void.

e) Templates except *AnyElementsOrNone* listed in the permutation shall obey the present template restriction (see clause 15.8).

Examples

EXAMPLE 1:

```
type record of integer MySequenceOfType;

template MySequenceOfType mw_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 mw_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 mw_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 mw_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 mw_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 mw_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 mw_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 mw_myInt1 := (1,2,3);
template integer mw_myInt2 := (1,2,?);
template integer mw_myInt3 := ?;
template integer mw_myInt4 := *;

template MySequenceOfType mw_myTemplate10 := { permutation (mw_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 mw_myTemplate11 := { permutation (mw_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 mw_myTemplate12 := { permutation (mw_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 mw_myTemplate13 := { permutation (mw_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 mw_myTemplate14 := { permutation (mw_myInt3, 2, ? ), 5 };
// matches any sequence of 4 integers that ends with 5 and contains 2 at least once in
// other positions

template MySequenceOfType mw_myTemplate15 := { permutation (mw_myInt4, 2, * ), 5 };
// matches any sequence of integers that ends with 5 and contains 2 at least once in
// other positions

template MySequenceOfType mw_myTemplate16 := { permutation (2, 2, 3 ), 5 };
// matches any sequence of integers of length 4 that ends with 5 and contains 2 in
// two other positions and 3 in the remaining position
```

EXAMPLE 2:

```
type record of integer RoI;
template RoI mw_roI1 := {1, 2, *};

template RoI mw_roI2 := {permutation(0, all from mw_roI1), 4, 5};
// results in {permutation(0, 1, 2, *), 4, 5}
```

B.1.4 Matching attributes of values

B.1.4.0 General

The following attributes may be associated with matching mechanisms.

B.1.4.1 Length restrictions

The **`length`** restriction attribute is used to restrict the length of string values matching the template or the number of elements in a **`set of`**, **`record of`** or array structure.

It can also be used in conjunction with the **`ifpresent`** matching attribute. The syntax for **`length`** can be found in clause 6.2.3.

NOTE: When the **`length`** attribute is used with a template list, elements of the list may be disabled by the attribute.

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.

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.

Restrictions

a) The length restriction shall be used only as an attribute of the following matching mechanisms: template list, complemented template list, *AnyValue*, *AnyValueOrNone*, *AnyElement*, *AnyElementsOrNone*, superset, subset, and pattern.

b) It shall not be used directly with templates and template fields produced by concatenation (see clause 15.11). If the length of a template or template field produced by concatenation is wished to be restricted, the concatenation shall be enclosed into a pair of parentheses.

c) The boundaries of the length restriction 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.

d) The length specifications for the template shall not conflict with the length for restrictions (if any) of the corresponding type.

e) 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.

Examples

```
// Given the message type definition
type record MyMessageType3
{
    record of integer   field1,
    charstring          field2,
    charstring          field3,
    charstring          field4
}

template MyMessageType3 mw_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).

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.

Restrictions

a) This *IfPresent* indicator shall be used only for matching mechanisms in templates of any type as a whole or for optional fields of **set** or **record** templates.

b) At the time of matching during a receiving operation, it shall be applied to optional fields of **record** and **set** templates only.

Examples

```
type record MyMessageType2
{
  integer           field1,
  MyRecordofType        field2  optional,
  boolean           field3  optional
}

template MyMessageType2 mw_myMessageTemplate:=
{
    :
    field2 := { 1, 2, 3 } ifpresent,    // matches { 1, 2, 3 } if not omitted
    :
}

template MyMessageType mw_myMessageTemplate2:=
{
  field1 := 1 ifpresent,               // causes an error as field1 is mandatory
  :
}

template MyRecordofType mw_myRecofTemplate := { 1, 2, 3 } ifpresent; // this assignment is
allowed

template boolean mw_myBoolTemplate := true ifpresent; // this assignment is also allowed

template MyMessageType2 mw_myMessageTemplate3:=
{
  field1 := 42,
```

```
  field2 := mw_myRecofTemplate,
    // if field2 is not absent, it matches the value { 1, 2, 3 }
  field3 := mw_myBoolTemplate
    // if field3 is not absent, it matches the value true
}

v_mybooleanVar := match ({},mw_myRecofTemplate);
  // mayches and returns true

v_mybooleanVar := match ({42,omit,true},mw_myMessageTemplate3);
  // matches and returns true
```

NOTE: *AnyValueOrNone* has exactly the same meaning as **? ifpresent**.

B.1.5 Matching character pattern

B.1.5.0 General

Character patterns can be used in templates to define the format of a required character string to be received. Character patterns can be used to match **charstring** and **universal charstring** values. In addition to literal characters, character patterns allow the use of meta-characters (e.g. ? and * within a character pattern means matching any character and any number of any character respectively).

EXAMPLE 1:

```
template charstring mw_myTemplate:= pattern "ab??xyz*0";
```

This template will 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 mw_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 in a set expression. It allows to specify a range of characters; see clause B.1.5.1 for more details
^	Has a metacharacter meaning in a set expression. It causes 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} or \q{Uxxxx, Uxxx ….}	Match one or more universal character. Both the quadruple and the USI-like syntaxes specified in clause 6.1.1 can be used
{reference}	Insert the referenced user defined string and interpret it as a regular expression. See clause B.1.5.2 for more details
{\reference}	Insert the referenced user defined string and interpret it as a set of literals. See clause B.1.5.2 for more details
\N{reference}	Matches a single character from the (sub)set of characters denoted; see clause B.1.5.4 for more details

Metacharacter	Description
\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 Recommendation ITU-T T.50 [4])
\n	Match any of the following C0 control characters: LF(10), VT(11), FF(12), CR(13) (see Recommendation ITU-T T.50 [4]) (jointly called newline characters, see clause A.1.5.1)
\r	Match the C0 control character CR (see Recommendation ITU-T 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 Recommendation ITU-T 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). See clause B.1.5.3 for more details
+	Match the preceding expression one or several times (postfix); the same as #(1,). See clause B.1.5.3 for more details
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 inTTCN-3 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.	

The symbols that can appear as lexical marks in metacharacter definitions are called metacharacter symbols.They include the following characters: "#", "(", ")", "*", "+", "-", "?", "[", "\", "]", "^", "{","|","}". When any of the metacharacter symbols are present in a pattern, but do not form a valid metacharacter, they retain their literal value.

NOTE: This rule assures that no format error can occur during pattern template instantiation. However, errors caused by invalid references can still appear (see clauses B.1.5.2 and B.1.5.4 for more details).

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 mw_myTemplate:= pattern "ab?\?" & "xyz*"; // results in the same pattern as
                                                              // in example 2
```

Pattern definitions may contain references to values or templates. The referred value or template shall be of the charstring or universal charstring type and it shall contain either a specific value or pattern. When the referenced template contains a pattern, the character pattern definition of this pattern is used as a fragment for creating the new pattern.

EXAMPLE 4:

```
template charstring mw_template1 := "ab?";
template charstring mw_template2 := pattern "?xyz*0";
template charstring mw_template3 := ?;
template charstring mw_template4 := pattern mw_template1 & mw_template2;
    // the same template as in example 1, i.e. pattern "ab??xyz*0"
template charstring mw_template5 := pattern mw_template3
    // produces as error as mw_template3 doesn't contain a value or pattern
```

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 "]" retains its literal value as no related opening bracket precedes it in the pattern. The pattern will match character strings containing two elements, with the first element equal to "a", "b", "[" or anything in the range "r"-"z" and the second character equal to "]".

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 mw_regExp1:= pattern "[a-z]";  //this will match any character from a to z

template charstring mw_regExp2:= pattern "[^a-z]"; //this will match any character except a to z

template charstring mw_regExp3:= pattern "[AC-E][0-9][0-9][0-9]YKE";

// mw_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, containing either a character string value or pattern matching. The reference shall be enclosed within the "{" "}" characters and reference shall resolve a compatible character string type. The opening bracket can be optionally followed by a backslash.

If the backslash character is missing, the referenced character string or pattern shall be inserted into the pattern being constructed and 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).

If the backslash character is present, the referenced item shall contain a character string value in this case. The character string is inserted into the pattern being constructed so that it all characters contained in it can keep their literal value (i.e. all metacharacter symbols are automatically escaped).

If the reference cannot be resolved or if the referenced symbol does not fulfil the requirements set by this clause, an error shall be generated.

EXAMPLE 1:

```
const charstring c_myString:= "ab?";

template charstring mw_myTemplate:= pattern "{c_myString}";
//matches any character string that consists of the characters "ab" followed by any character

template charstring mw_myTemplate2:= pattern "{\c_myString}";
//resolves into pattern "ab\?" and matches the string"ab?" only

template universal charstring mw_myTemplate3:= pattern "{c_myString}de\q{1, 1, 13, 7}";
//matches any universal 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 c_myConst2 := "ab";
template charstring mw_regExp1 := pattern "{c_myConst2}";
   // matches the string "ab"
template charstring mw_regExp1a := pattern c_myConst2;
   // the same as above, matches the string "ab"
template charstring mw_regExp2 := pattern "{mw_regExp1}{mw_regExp1}";
   // matches the string "abab"
template charstring mw_regExp2a := pattern "{mw_regExp1}" & "{mw_regExp1}";
   // the same as above, matches the string "abab"
template charstring mw_regExp2b := pattern mw_regExp1 & mw_regExp1;
   // the same as above, matches the string "abab"
template charstring mw_regExp3 := pattern "c{mw_regExp2}d";
   // matches the string "cababd"

template charstring mw_regExp4 := pattern "{mw_reg";
template charstring mw_regExp5 := pattern "Exp1}";
template charstring mw_regExp6 := pattern "{mw_regExp4}{mw_regExp5}";
   // matches the string "{mw_regExp1}" only (i.e. shall not be handled as a reference
   // expression after insertion)
template charstring mw_regExp7 := pattern "{mw_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 m_ref0:= "My String";
template charstring m_ref1:= "{m_re";
template charstring m_ref2:= "f0}";
template charstring m_ref3:= "{m_ref1}{m_ref2}";
   //this matches "{m_ref0}"
   //i.e. there is no further dereferencing
   //as m_ref1 and m_ref2 do not contain a reference

template charstring m_ref4:= "{m_ref0}";
template charstring m_ref5:= "";
template charstring m_ref6:= "{m_ref4}{m_ref5}";
   //this matches "My String" – here m_ref0 is dereferenced, because m_ref4 contains
   //the reference expression {m_ref0} with the reference m_ref0
```

EXAMPLE 4:

```
type record MyRecordType {
    integer i,
    charstring c
}
```

```
const MyRecordType c_referencedRecord:= {1,"this"}
const charstring c_referencedConstant := c_referencedRecord.c;
template charstring m_referencingPattern := pattern "{c_referencedConstant}"
   //this matches "this" as the c_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 shall be matched at least n times but not more than m times.

The metacharacter postfix "#(n,)" specifies that the preceding expression shall 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 shall be matched exactly n times (they are equivalent to "#(n, n)"). In the form "#n" n shall be a single digit.

The forms "#(,)" and "#()" are shorthand notations for "#(0,)", i.e. matches the preceding expression any number of times.

The metacharacter postfix "+" denotes that the preceding expression shall be matched at least 1 time (equivalent to "#(1,)").

EXAMPLE:

```
template charstring mw_regExp4:= pattern "[a-z]#(9, 11)"; //match at least 9 but no more than 11
                                                          // characters from a to z
template charstring mw_regExp5a:= pattern "[a-z]#(9)";    // match exactly 9
                                                          // characters from a to z
template charstring mw_regExp5b:= pattern "[a-z]#9";      // match exactly 9
                                                          // characters from a to z
template charstring mw_regExp6:= pattern "[a-z]#(9, )";   // match at least 9
                                                          // characters from a to z
template charstring mw_regExp7:= pattern "[a-z]#(, 11)";  // match no more than 11
                                                          // characters from a to z
template charstring mw_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.

If the reference cannot be resolved or if the referenced symbol is anything else than a template, constant, variable, formal parameter or module parameter containing a character string of length 1, an error shall be generated.

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 MyCharRangeType ("a".."z");
type charstring MyCharListType ("a", "z");
const MyCharRangeType c_myCharR := "r";

template charstring mw_myTempPatt1 := pattern "\N{c_myCharR}";
// mw_myTempPatt1 shall match the string "r" only
```

```
template charstring mw_myTempPatt2 := pattern "\N{MyCharRange}";
// mw_myTempPatt2 shall match any string containing a single character from a to z

template MyCharRangeType mw_myTempPatt3 := pattern "\N{MyCharList}";
// mw_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).

B.1.5.6 Case insensitive pattern matching

When the "**@nocase**" modifier is used after the pattern keyword, the matching is evaluated in a case insensitive way, i.e. at positions, where without the "**@nocase**" modifier a small letter alphabethical character would be matched, with the "**@nocase**" modifier also capital letter counterpart - but only that - shall be accepted. For example, at positions where the pattern matches the character đ (latin small letter d with stroke), also its counterpart Đ（latin capital letter d with stroke）shall be accepted, but the similarly looking graphical characters Ð (latin capital letter eth) and Ɖ (latin capital letter african d) shall not.

EXAMPLE 1:

```
template charstring mw_myTemplateNoCase:= pattern @nocase "ab??xyz*0";
//This template would match any character string that start with the characters "ab" or "Ab"
//or "aB" or "AB", followed by any two characters, followed by the characters "xyz" or "Xyz"
//or "xYz" or "xyZ" or "XYz" or "xYZ" or "XyZ" or "XYZ", followed by any number of any
//characters (including any number of "0"-s) before the closing character "0".
```

When referencing a pattern from inside another pattern (see clause B.1.5.2), the case sensitivity property of the referenced pattern is not inherited. I.e. - after dereferencing, possibly recursively - only the resulting string part of the referenced pattern is inserted into the referencing pattern. The whole resulting pattern is always evaluated according to the case-sensitivity of the referencing pattern.

EXAMPLE 2:

```
const charstring c_myString:= "ab?";

template charstring mw_myTemplate:= pattern @nocase "{c_myString}";
//matches any character string that consists of the characters "ab" or "Ab" or "aB" or "AB",
// followed by any character

template universal charstring mw_myTemplate3:= pattern "{mw_myTemplate}de\q{1, 1, 13, 7}";
//matches 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 (.
```

Annex C (normative): Predefined TTCN-3 functions

C.0 General exception handling procedures

This annex defines the TTCN-3 predefined functions.

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 runtime error. Which error situation causes compile-time and which one runtime error is a tool implementation option.

C.1 Conversion functions

C.1.1 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.1.2 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.1.3 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.1.4 Integer to enumerated

```
int2enum ( in integer inpar, out Enumerated_type outpar)
```

This function converts an integer value into an enumerated 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, Weekend(6..7)
};

type enumerated MySecondEnumType {
    Saturday(-3), Sunday (0), Monday
};

//within a dynamic language element:
var MyFirstEnumType v_firstEnum := Tuesday;
var MySecondEnumType v_secondEnum := Sunday;

int2enum(0, v_firstEnum)    // v_firstEnum == Monday
int2enum(1, v_secondEnum)   // v_secondEnum == Monday
int2enum(6, v_firstEnum)    // v_firstEnum == Weekend(6)
```

C.1.5 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.1.6 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.1.7 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.1.8 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.1.9 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.1.10 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.1.11 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 Recommendation ITU-T 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.1.12 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.1.13 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 runtime error. However, this is out of the scope of the present document.

The general error causes in clause 16.1.2 apply.

C.1.14 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.1.15 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.1.16 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.1.17 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 runtime error. However, this is out of the scope of the present document.

The general error causes in clause 16.1.2 apply.

C.1.18 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.1.19 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.1.20 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.1.21 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 runtime error. However, this is out of the scope of the present document.

The general error causes in clause 16.1.2 apply.

C.1.22 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.1.23 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.1.24 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.1.25 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 Recommendation ITU-T 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.1.26 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 runtime 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.1.27 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.1.28 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.2.1 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 in the charstring value "01D7"
```

C.1.29 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 runtime 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.1.30 Enumerated to integer

```
enum2int(in Enumerated_type inpar) return integer
```

This function accepts an enumerated value and returns the **integer** value associated to the enumerated value (see also clause 6.2.4). The actual parameter passed to inpar always shall be a typed object (see clause 6.2.4 and the definition "type context" in clause 3.1).

The general error causes in clause 16.1.2 apply.

EXAMPLE:

```
type enumerated MyFirstEnumType {
    Monday, Tuesday, Wednesday, Thursday, Friday, Weekend(5..6)
};

type enumerated MySecondEnumType {
    Saturday(-3), Sunday (0), Monday
};

//within a dynamic language element:
var MyFirstEnumType v_firstEnum := Monday;
var MySecondEnumType v_secondEnum := Monday;

enum2int(v_firstEnum)  // returns 0
enum2int(v_secondEnum) // returns 1

v_firstEnum := Wednesday;
v_secondEnum := Saturday;
enum2int(v_firstEnum)  // returns 2
enum2int(v_secondEnum) // returns -3
```

```
v_firstEnum := Friday;
v_secondEnum := Sunday;
enum2int(v_firstEnum)  // returns 4
enum2int(v_secondEnum) // returns 0
v_firstEnum := Weekend(6);
enum2int(v_firstEnum)  // returns 6
```

C.1.31 Octetstring to universal character string

```
oct2unichar(in octetstring invalue, in charstring string_encoding := "UTF-8")
  return universal charstring
```

This function converts an **octetstring** invalue to a **universal charstring** by use of the given string_encoding. The octets are interpreted as mandated by the standardized mapping associated with the given string_encoding and the resulting characters are appended to the returned value. If the optional string_encoding parameter is omitted, the default value "UTF-8".

The following values (see ISO/IEC 10646 [2]) are allowed as string_encoding actual parameters (for the description of the codepoints see clause 27.5):

a) "UTF-8"

b) "UTF-16"

c) "UTF-16LE"

d) "UTF-16BE"

e) "UTF-32"

f) UTF-32LE"

g) "UTF-32BE"

The invalue parameter shall not include the optional signature (see clause 10 of ISO/IEC 10646 [2], also known as byte order mark).

In case of "UTF-16" and "UTF-32" big-endian ordering shall be used (as described in clauses 10.4 and 10.7 of ISO/IEC 10646 [2]).

The general error causes in clause 16.1.2 apply.

EXAMPLE:

```
oct2unichar('C384C396C39CC3A4C3B6C3BC'O) = "ÄÖÜäöü";
oct2unichar('00C400D600DC00E400F600FC'O,"UTF-16BE") = "ÄÖÜäöü";
oct2unichar('C400D600DC00E400F600FC00'O,"UTF-16LE") = "ÄÖÜäöü";
```

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

C.1.32 Universal character string to octetstring

```
unichar2oct(in universal charstring invalue, in charstring string_encoding := "UTF-8")
return octetstring
```

This function converts a **universal charstring** invalue to an **octetstring**. Each octet of the octetstring will contain the octets mandated by mapping the characters of invalue using the standardized mapping associated with the given string_encoding in the same order as the characters appear in inpar. If the optional string_encoding parameter is omitted, the default value "UTF-8".

The following values (see ISO/IEC 10646 [2]) are allowed as string_encoding actual parameters (for the description of the UCS encoding scheme see clause 27.5):

a) "UTF-8"

b) "UTF-16"

c) "UTF-16LE"

d) "UTF-16BE"

e) "UTF-32"

f) "UTF-32LE"

g) "UTF-32BE"

The general error causes in clause 16.1.2 apply.

EXAMPLE:

```
unichar2oct("ÄÖÜäöü") = 'C384C396C39CC3A4C3B6C3BC'O;
unichar2oct("ÄÖÜäöü","UTF-16BE") = '00C400D600DC00E400F600FC'O;
unichar2oct("ÄÖÜäöü","UTF-16LE") = 'C400D600DC00E400F600FC00'O;
```

C.1.33 Value or template to universal charstring

```
any2unistr(in template any_type invalue) return universal charstring
```

This function converts the content of a value or template to a single **universal charstring**. The resulting **universal charstring** is the same as the string produced by the log operation containing the same operand as the one passed to the **any2unistr** function. The value or template passed as a parameter to the **any2unichar** function may be uninitialized, partially or completely initialized.

The general error causes in clause 16.1.2 apply.

EXAMPLE:

```
var integer v_int1 := 5, v_int2;
var template integer mw_int1 := ?;
var template integer mw_int2 := -1 ifpresent;
var universal charstring v_chr1, v_chr2, v_chr3, v_chr4;
v_chr1 := any2unistr(v_int1); // after the assignment v_chr1 will be "5"
v_chr2 := any2unistr(v_int2); // after the assignment v_chr2 will be "UNINITIALIZED"
v_chr3 := any2unistr(mw_int1); // after the assignment v_chr3 will be "?"
v_chr4 := any2unistr(mw_int2); // after the assignment v_chr3 will be "-1 ifpresent"
```

C.2 Length/size functions

C.2.1 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. The units of length for each string type are defined in table 4 in the present document.

For values or templates of **record of** or **set of** type, the value to be returned is the maximum of the minimal length restriction value of the type, or 0 for types with no minimal length restriction, and the index of the last initialized element plus 1.

The length value returned in case of length restricted string or list type shall be at least the minimum length according to the type definition. In particular, the length of a fixed length **record of** or **set of** value will always be the fixed length according to the type definition. For array values or templates, the value to be returned is the fixed length of the corresponding **record of** type.

NOTE 1: As **in** formal parameters does not allow passing in uninitialized values or templates, even in these cases `inpar` will be at least partially initialized.

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. If inpar contains uninitialized elements, each of them shall be counted as 1 element, i.e. they shall be matched as if they contained the "?" (*AnyElement* inside value) matching character in case of binary strings or as if they were a "?" (Match any character) character pattern for textual strings.

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. If inpar contains uninitialized elements, each of them shall be counted as 1 element, i.e. they shall be matched as if they contained the "?" (*AnyElement* inside value) matching character.

NOTE 2: 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 when 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 3: On real test systems the length calculation of inpar may lead to an overflow problem that causes compile time or runtime error. However, this is out of the scope of the present document.

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 v_myListVar := { 0, 1, -, 2, - };

lengthof(v_myListVar);
// returns 4 without respect to the fact, that the element v_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 mw_roI1 := { 1, permutation(2, 3), ? }
template RoI mw_roI2 := {1, *, (2, 3) }
template RoI mw_roI3 := { 1, *, 10 } length(5)
template RoI mw_roI4 := { 1, 2, 3, * } length(1..2)
template RoI mw_roI5 := { 1, 2, 3, * } length(1..3)

lengthof (mw_roI1) // returns 4

lengthof (mw_roI2) // shall cause an error

lengthof (mw_roI3) // returns 5

lengthof (mw_roI4) // shall cause an error

lengthof (mw_roI5) // returns 3
```

C.2.2 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 m_myTemplate :=
    { field1 := omit,
      field2 := 5
    };

sizeof(m_myTemplate);   // returns 1

type set S {
  integer f1,
  bitstring f2 optional,
  charstring f3 optional
}

template S mw_s1 := { f1 := (0..99), f2 := omit, f3 := ? }
template S mw_s2 := { f3 := *, f1 := 1, f2 := '00'B ifpresent }
template S mw_s3 := ({ f1 := 1, f2 := omit, f3 := "ABC" },
                     { f1 := 2, f3 := omit, f2 := '1'B },
                     { f3 := omit, f1 := 3, f2 := '1?1'B }
                    )
template S mw_s4 := ?

sizeof(mw_s1) // returns 2
sizeof(mw_s2) // shall cause an error
sizeof(mw_s3) // returns 2
sizeof(mw_s4) // shall cause an error
```

C.3 Presence checking functions

C.3.1 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 data object reference fulfils the (present) template restriction as described in clause 15.8;
- the value **`false`** otherwise.

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`**`(m_myTemplate)` returns `true`:

- `m_myTemplate` can safely be assigned to a non-optional field of the type of the template in a template variable;
- `m_myTemplate` can safely be used as an actual template(present) parameter or assigned to a variable of kind template(present).

When applying the **`ispresent`** function to a semantically correct data object reference (see table 14 in clause 16.1.2), it shall never result in an error, even if using the reference would normally cause a runtime error when being used e.g. in an expression.

EXAMPLE:

```
// Given
type record MyRecord
  {
    record {
      boolean innerField1 optional,
      integer innerField2 optional,
      MyRecord innerField3 optional
    } field1 optional,
      integer field2
    }

var MyRecord v_myRecord := { field1 := {}, field2 := 5 }
// type of field1 is record with fields, therefore field1 remains uninitialized
// after this assignment (no value is assigned to any of the fields of vl_MyRecord.field1)

ispresent(v_myRecord.field1) // returns false


v_myRecord.field1 := omit

ispresent(v_myRecord.field1) // returns false
// and therefore, v_myRecord.field1.innerField1 is an inaccessible reference

ispresent(v_myRecord.field1.innerField3.field2) // shall return false because innerField3 is
// unintialized and therefore, v_myRecord.field1.innerField3.field2 is an
// inaccessible reference

ispresent(v_myRecord.field1.innerField1) // shall return false because field1 is omitted

var template MyRecord v_myRecordT := { field1 := ?, field2 := 5 }

ispresent(v_myRecordT.field1) // returns true

ispresent(v_myRecordT.field1.innerField1) // returns false 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 mw_t1 := {f1 := 1, f2 :=(2 .. 4) }
template R mw_t2 := { f1 := omit, f2 := (5, 7) ifpresent }
template R mw_t3 := {f1 := *, f2 :=? }
```

```
ispresent(mw_t1.f1) // returns true

ispresent(mw_t1.f2) // returns true

ispresent(mw_t2.f1) // returns false

ispresent(mw_t2.f2) // returns false

ispresent(mw_t3.f1) // returns false

ispresent(mw_t3.f2) // returns true
```

C.3.2 The IsChosen function

```
ischosen(in template any_union_type_field inpar) return boolean
```

This function is allowed for templates of all data types that are a union-field-reference or a type alternative of an **anytype**. This function returns:

- the value **true** if and only if the data object reference specifies the variant of the **union** type or the type alternative of the **anytype** that is actually selected for the given data object;
- in all other cases **false**.

The function **ischosen** is applicable to templates of **union** types or of **anytype** containing a specific value or a value list. It returns **true** if all the values matched by `inpar` have the given alternative selected. The result is **false** if there is another alternative of the **union** type or **anytype** on which **ischosen** would return true.

NOTE: Please note that in case of **anytype**-s, no type compatibility is considered when determining the selected alternative; i.e. at the evaluation only the exact type chosen for the given value will satisfy the above criteria.

The application of the **ischosen** function to a semantically correct data object reference shall never result in an error, even if using the reference would normally cause a runtime error when being used e.g. in an expression.

EXAMPLE 1: Using **ischosen** for **union** types

```
// inside module M:
type union U { integer f1, octetstring f2 }
template U m_u1 := {f1 := 1}
template U mw_u2 := {f2 := ?}
template U mw_u3 := ?
template U m_u4 := ({ f1 := 2 }, {f2 := 'AB'O })
template U mw_u5 := ({ f2 := '12?'O }, { f2 := '*34'O length(2) })

ischosen(m_u1.f1) // returns true

ischosen(m_u1.f2) // returns false

ischosen(mw_u2.f1) // returns false

ischosen(mw_u2.f2) // returns true

ischosen(mw_u3.f1) // returns false

ischosen(mw_u3.f2) // returns false

ischosen(m_u4.f1) // returns false

ischosen(m_u4.f2) // returns false

ischosen(mw_u5.f1) // returns false

ischosen(mw_u5.f2) // returns true


type record R { U u optional }
template R m_r1 := { omit }

ischosen(m_r1.u.f1) // returns false
```

EXAMPLE 2: Using **ischosen** for **anytype**

```
template anytype mw_a1 := { U := mw_u5 }
template anytype mw_a2 := { M.U := { f1 := m_u1.f1 } }
ischosen(mw_a1.U)        // returns true
ischosen(mw_a1.M.U)      // returns true
ischosen(mw_a1.integer)  // returns false
ischosen(mw_a2.U)        // returns true
```

EXAMPLE 3:

```
// Given
type union MyUnion
    {   PDU_type1   p1,
        PDU_type2   p2,
        PDU_type    p3
    }

// and given that mw_myPDU is a template of MyUnion type
// and v_receivedPDU is also of MyUnion type
// then
myPort.receive(mw_myPDU) -> value v_receivedPDU
ischosen(v_receivedPDU.p2)
// returns true if the actual instance of mw_myPDU carries a PDU of the type PDU_type2
```

C.3.3 The IsValue function

```
isvalue(in template any_type inpar) return boolean;
```

This function is allowed for templates of all data types, component and address types and default values. The function shall return **true**, if `inpar` is completely initialized and resolves to a specific value. If `inpar` is of **record** or **set** type, omitted optional fields shall be considered as initialized, i.e. the function shall also return true if optional fields of `inpar` are set to omit. The function shall return **false** otherwise.

The **null** value assigned to default and component references shall be considered as concrete values.

The application of the **isvalue** function to a semantically correct data object reference shall never result in an error, even if using the reference would normally cause a runtime error when being used e.g. in an expression.

EXAMPLE 1: Simple types

```
template charstring m_char0 := "ABCD";  //template containing a specific value matching
template charstring m_char1 := "AB?D";  //template containing a specific value matching
                                        //note, that "?" is not a matching symbol in this case
template charstring mw_char2 := pattern "ABCD"; // a pattern matching a single value only
template charstring mw_char3 := pattern "AB?D"; // pattern matching
template charstring m_char4 := ("ABCD"); // template containing a specific value (expression)
template charstring mw_char5 := ("ABCD","EFGH"); // a value list matching a single value only

isvalue(m_char0);  // shall return true
isvalue(m_char1);  // shall return true
isvalue(mw_char2); // shall return false
isvalue(mw_char3); // shall return false
isvalue(m_char4);  // shall return true similarly to e.g. isvalue((2)) shall return true
isvalue(mw_char5); // shall return false
```

EXAMPLE 2: Special types

```
var default v_default := null;
isvalue(v_default); // shall return true
```

EXAMPLE 3: Record/set types

```
type record MyRec {
    integer f1 optional,
    integer f2 optional
}

var MyRec v_myRec;
var template MyRec v_myRecT;

isvalue(v_myRec);      // shall return false
```

```
isvalue(v_myRecT);    // shall return false

v_myRec   := { f1 := 5, f2 := omit }
v_myRecT  := { f1 := ?, f2 := 5 }

isvalue(v_myRec);      // shall return true
isvalue(v_myRec.f2);  // shall return false;
isvalue(v_myRecT);    // shall return false
isvalue(v_myRecT.f1); // shall return false
isvalue(v_myRecT.f2); // shall return true

v_myRecT.f2 := omit;

isvalue(v_myRecT.f2); // shall return false
```

EXAMPLE 4: Union types

```
type union MyUnion {
    integer ch1,
    integer ch2
}

template MyUnion m_myUnion := { ch1 := 5 }
template MyUnion mw_myUnion := { ch1 := ? }


isvalue(m_myUnion);      // shall return true
isvalue(mw_myUnion);     // shall return false
isvalue(mw_myUnion.ch1); // shall return false
// note, this is different from ischosen(mw_myUnion.ch1) as isvalue checks the content of the
// choice ch1, while ischosen is checking if ch1 has been selected or not
isvalue(mw_myUnion.ch2); // shall return false
```

EXAMPLE 5: Nested types

```
type record MyRecord {
    MyUnion u optional
}

template MyRecord m_myRecord  := { u := m_myUnion }
template MyRecord mw_myRecord := { u := mw_myUnion }
template MyRecord m_myRecord2 := { u := omit }

isvalue(m_myRecord.u.ch1);  // shall return true
isvalue(mw_myRecord.u.ch1); // shall return false
isvalue(mw_myRecord.u.ch2); // shall return false
isvalue(m_myRecord.u.ch2);  // shall return false
```

C.3.4 The IsBound function

```
isbound(in template any_type inpar) return boolean;
```

This function is allowed for templates of all data types. The function shall return **`true`**, if `inpar` is at least partially initialized. If `inpar` is of a **`record`** or **`set`** type, omitted optional fields shall be considered as initialized, i.e. the function shall also return **`true`** if at least one optional field of `inpar` is set to **`omit`**. The function shall return **`false`** otherwise. Inaccessible fields (e.g. non-selected alternatives of **`union`** types, subfields of omitted record and set types or subfields of non-selected union fields) shall be considered as uninitialized, i.e. isbound shall return for them **`false`**.

The **`null`** value assigned to default and component references shall be considered as concrete values.

The application of the **`isbound`** function to a semantically correct template reference shall never result in an error, even if using the reference would normally cause a runtime error when being used e.g. in an expression.

EXAMPLE 1: Simple types

```
var template charstring v_char;
isbound(v_char);           // shall return false as v_char is uninitialized;

v_char := "AB?D";          // template containing a specific value
isbound(v_char);           // shall return true

v_char := pattern "AB?D";  //template containing a pattern matching
isbound(v_char);           // shall return true
```

EXAMPLE 2: Special types

```
var default v_default := null;
isbound(v_default);         // shall return true
```

EXAMPLE 3: Record/set types

```
type record MyRec {
    integer f1,
    MyRec f2 optional
}

var MyRec v_myRec;
isbound(v_myRec);           // shall return false

v_myRec.f2 := omit;
isbound(v_myRec);           // shall return true as v_myRec is partially initialized,
                            // field f2 is set to omit

v_myRec  := { f1 := 5, f2 := omit }
isbound(v_myRec);           // shall return true as v_myRec is completely initialized
isbound(v_myRec.f2.f1);     // shall return false as v_myRec.f2.f1 is inaccessible
isbound(v_myRec.f1/0);      // shall cause an error already during evaluating the argument
                            // as division by zero is not allowed

type union MyUnion {
    integer ch1,
    MyRec   ch2
}

var template MyUnion v_myUnion;
isbound(v_myUnion);        // shall return false, as v_myUnion is uninitialized
isbound(v_myUnion.ch1);    // shall return false, as alternative ch1 is uninitialized

v_myUnion := { ch1 := 5 };
isbound(v_myUnion);        // shall return true
isbound(v_myUnion.ch1);    // shall return true
isbound(v_myUnion.ch2);    // shall return false as the ch2 alternative is not selected
isbound(v_myUnion.ch2.f1); // shall return false as the field f1 is inaccessible
isbound(v_myUnion.ch1/0);  // shall cause an error already during evaluating the argument
                           // as division by zero is not allowed
```

C.3.5 Matching mechanism detection

```
istemplatekind (in template any_type invalue, in charstring kind) return boolean
```

This function allows to examine if a template contains a certain kind of the matching mechanisms.

If the matching mechanism kind enquired is matching a specific value (clause B.1.1), a matching mechanism instead of values (clause B.1.2) or matching character pattern (clause B.1.5), the function shall return **`true`** if the content of the invalue parameter is of the same kind.

If the matching mechanism kind enquired is a matching mechanism inside values (clause B.1.3), the function shall return **`true`** if the template in the invalue parameter contains this kind of matching mechanism on the first level of nesting.

If the matching mechanism kind enquired is a matching attribute (clause B.1.4), the function shall return **`true`** if the template in the invalue parameter has this kind of matching attribute attached to it directly (i.e. it doesn't count if the attribute is attached to a field of invalue at any level of nesting).

In all other cases the function returns **`false`**.

The `kind` parameter shall be one of the strings listed in table C.1.

Table C.1: Allowed values of kind parameter

Value of kind parameter	Searched matching mechanism	
	Name	Clause reference
"value"	Specific value	B.1.1
"list"	Template list	B.1.2.1
"complement"	Complemented template list	B.1.2.2
"AnyValue", "?"	Any value	B.1.2.3
"AnyValueOrNone", "*"	Any value or none	B.1.2.4
"range"	Value range	B.1.2.5
"superset"	SuperSet	B.1.2.6
"subset"	SubSet	B.1.2.7
"omit"	Omit	B.1.2.8
"decmatch"	Matching decoded content	B.1.2.9
"AnyElement"	Any element	B.1.3.1
"AnyElementsOrNone"	Any number of elements or none	B.1.3.2
"permutation"	Permutation	B.1.3.3
"length"	Length restriction	B.1.4.1
"ifpresent"	The *IfPresent* indicator	B.1.4.2
"pattern"	Matching character pattern	B.1.5

NOTE: Clause E.2.2.5 includes the type definition TemplateKind and a constant for each of the allowed values of the kind parameter. It is recommended to use the istemplatekind function in combination with this type and these constants to ease the checking of correct usage and to improve the readability of test specs.

Restrictions

In addition to the general error causes given in clause 16.1.2, the following restrictions apply:

a) Calling the `istemplatekind` function with a different second parameter than stated in table C.1 shall lead to an error.

EXAMPLE:

```
type record of integer RoI;
...
var template integer v_t1 := ?, v_t2 := (0,1,2) ifpresent;
var template RoI v_t3:= { permutation(1, 2, 3), ? };
var boolean v_res;
...
v_res := istemplatekind(v_t1, "AnyValue");       // true
v_res := istemplatekind(v_t1, "AnyValueOrNone"); // false
v_res := istemplatekind(v_t2, "complement");     // false
v_res := istemplatekind(v_t2, "list");           // true
v_res := istemplatekind(v_t2, "ifpresent");      // true
v_res := istemplatekind(v_t3, "permutation");    // true
v_res := istemplatekind(v_t3, "AnyElement");     // true
```

C.4 String/list handling functions

C.4.1 The Regexp function

```
regexp [@nocase] (
  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 first matches the parameter `inpar` (or in case `inpar` is a template, its value equivalent)against the `expression` in the second parameter according to the pattern matching specified in clause B.1.5. 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 the **`@nocase`** modifier is present, this and all subsequent matchings shall be done in a case-insensitive way, as specified in clause B.1.5.6. If `inpar` is a literal (i.e. type is not explicitly given) the corresponding type shall be retrieved from the value contents.

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.

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. 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`.

NOTE: This function differs from other well-known regular expression matching implementations in that:

a) It shall match the whole inpar string instead of only a substring.

b) It starts counting groups from 0, while in some other implementations the first group is referenced by 1 and the whole substring matched by the expression is referenced by 0.

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 v_myInput := "   simple text for a regexp example   ";
var charstring v_myString;

v_myString := regexp(v_myInput,charstring:"?+(text)?+",0);
  // will return "text"

v_myString := regexp(v_myInput,charstring:"?+(text)?+",1);
  // causes an error as there is no group with index 1

v_myString := regexp(v_myInput,charstring:"(?+)(text)(?+)",0);
  // will return "   simple "

v_myString := regexp(v_myInput,charstring:"(?+)(text)(?+)",2);
  // will return " for a regexp example   "

v_myString := regexp(v_myInput,charstring:"((?+)(text)(?+))",0);
  // will return the whole inpar, i.e. "   simple text for a regexp example   "

v_myString := regexp(v_myInput,charstring:"(([ ]+)(text)(?+))",0);
  // will return an empty string as expression does not matches inpar

v_myString := regexp(v_myInput,universal charstring:"?+(text)?+",0);
  // will cause an error as inpar is of type charstring, while
  // expression is of type universal charstring

v_myInput := "      date: 2001-10-20 ;  msgno: 17; exp  ";
var template charstring v_myPattern :=
  pattern "([ \t]#(0,)date:[ \d\-]#(0,);[ \t]#(0,)msgno: (\d#(1,3)); (exp)#(0,1)) [ \t]#(0,)";
  // please note, that only the very first opening bracket and the bracket before "\d#(1,3)"
  // denotes groups; "#(0,)", "#(1,3)" and "#(0,1)" denotes matching the preceding expression
  // several time

v_myString := regexp(v_myInput, v_myPattern,0);
  // will return the input string but the whitespace at the end,
  // i.e. "      date: 2001-10-20 ;  msgno: 17; exp"
```

```
v_myString := regexp(v_myInput, v_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 f_regexp0(
  in template charstring p_inpar,
  in template charstring p_expression,
  in integer p_groupno)
return charstring {
    var template charstring v_extendedExpr := pattern "({p_expression})";
    return regexp(p_inpar, v_extendedExpr, p_groupno )
}
```

C.4.2 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). If `inpar` is a literal (i.e. type is not explicitly given) the corresponding type shall be retrieved from the value contents. 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 some 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 specific value 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.4.3 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`. If `inpar` is a literal (i.e. type is not explicitly given) the corresponding type shall be retrieved from the value contents. `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 some 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.5 Codec functions

C.5.1 The encoding function

```
encvalue(in template (value) any_type inpar,
         in universal charstring encoding_info := "",
         in universal charstring dynamic_encoding := "") 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 make any check on its correctness. The optional encoding_info parameter is used for passing additional encoding information to the codec and, if it is omitted, no additional information is sent to the codec.

The optional dynamic_encoding parameter is used for dynamic selection of **encode** attribute of the **inpar** value for this single **encvalue** call. The rules for dynamic selection of the **encode** attribute are described in clause 27.9.

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.5.2 The decoding function

```
decvalue(inout bitstring encoded_value,
        out any_type decoded_value,
         in universal charstring decoding_info := "",
         in universal charstring dynamic_encoding := "") 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. The optional decoding_info parameter is used for passing additional decoding information to the codec and, if it is omitted, no additional information is sent to the codec.

The optional dynamic_encoding parameter is used for dynamic selection of **encode** attribute of the decoded_value parameter for this single **decvalue** call. The rules for dynamic selection of the **encode** attribute are described in 27.9.

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. This value is also returned if the encoded_value parameter contains an unitialized value.
- 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.

C.5.3 The encoding to universal charstring function

```
encvalue_unichar(in template (value) any_type inpar,
                 in charstring string_serialization := "UTF-8",
                 in universal charstring encoding_info := "",
                 in universal charstring dynamic_encoding := "")
                 return universal charstring
```

The **encvalue_unichar** function encodes a value or template into a universal charstring. 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 universal charstring represents the encoded value of inpar, however, the TTCN-3 test system need not make any check on its correctness. If the optional string_serialization parameter is omitted, the default value "UTF-8" is used. The optional encoding_info parameter is used for passing additional encoding information to the codec and, if it is omitted, no additional information is sent to the codec.

The optional dynamic_encoding parameter is used for dynamic selection of **encode** attribute of the **inpar** value for this single encvalue_unichar call. The rules for dynamic selection of the **encode** attribute are described in clause 27.9.

The following values (see ISO/IEC 10646 [2]) are allowed as string_serialization actual parameters (for the description of the UCS encoding scheme see clause 27.5):

a) "UTF-8"

b) "UTF-16"

c) "UTF-16LE"

d) "UTF-16BE"

e) "UTF-32"

f) "UTF-32LE"

g) "UTF-32BE"

The serialized bitstring shall not include the optional signature (see clause 10 of ISO/IEC 10646 [2], also known as byte order mark).

In case of "UTF-16" and "UTF-32" big-endian ordering shall be used (as described in clauses 10.4 and 10.7 of ISO/IEC 10646 [2]).

The specific semantics of this function are explained by the following TTCN-3 definition:

```
function encvalue_unichar(in template(value) any_type inpar,
                          in charstring enc
                          in universal charstring encoding_info := "",
                          in universal charstring dynamic_encoding := "")
  return universal charstring {
    return oct2unichar(bit2oct(encvalue(inpar, encoding_info, dynamic_encoding)), enc);
}
```

```
The encvalue_unichar function first invokes the encvalue function in order to encode the value
passed in the inpar parameter to a bitstring. The bitstring is then converted to an octetstring by
the bit2oct function and subsequently to a universal charstring using the oct2unichar function. The
string_serialization parameter defines how the encoded octets (in fact the encoded bitstring
received from the codec) contain the characters. The universal charstring value is then returned as
the result of the encvalue_unichar function.
```

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).
- The given string encoding is not recognized.

C.5.4 The decoding from universal charstring function

```
decvalue_unichar(inout universal charstring encoded_value,
                 out any_type decoded_value,
                 in charstring string_serialization:= "UTF-8",
                 in universal charstring decoding_info := "",
                 in universal charstring dynamic_encoding := "")
                 return integer
```

The **decvalue_unichar** function decodes (part of) a universal charstring into a value. The test system shall suppose that a prefix of the universal charstring encoded_value represents an encoded instance of the actual type of decoded_value. The optional decoding_info parameter is used for passing additional decoding information to the codec and, if it is omitted, no additional information is sent to the codec.

The optional dynamic_encoding parameter is used for dynamic selection of **encode** attribute of the decoded_value parameter for this single decvalue_unichar call. The rules for dynamic selection of the **encode** attribute are described in clause 27.9.

If the decoding was successful, then the characters used for decoding 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. This value is also returned if the encoded_value parameter contains an unitialized value.
- The return value 2 indicates that decoding could not be completed as encoded_value did not contain enough bits.

If the optional string_serialization parameter is omitted, the default value "UTF-8" is used.

The following values (see ISO/IEC 10646 [2]) are allowed as string_serialization actual parameters (for the description of the UCS encoding scheme see clause 27.5):

a) "UTF-8"

b) "UTF-16"

c) "UTF-16LE"

d) "UTF-16BE"

e) "UTF-32"

f) "UTF-32LE"

g) "UTF-32BE"

The serialized bitstring shall not include the optional signature (see clause 10 of ISO/IEC 10646 [2], also known as byte order mark).

In case of "UTF-16" and "UTF-32" big-endian ordering shall be used (as described in clauses 10.4 and 10.7 of ISO/IEC 10646 [2]).

The semantics of the function can be explained by the following TTCN-3 function:

```
function decvalue_unichar (inout universal charstring encoded_value,
                           out any_type decoded_value,
                           in charstring string_encoding := "UTF-8"",
                           in universal charstring decoding_info := "",
                           in universal charstring dynamic_encoding := "") return integer {
    var bitstring v_str = oct2bit(unichar2oct(encoded_value, string_encoding));
    var integer v_result := decvalue(v_str, decoded_value, decoding_info, dynamic_encoding);
    if (v_result == 0) { // success
      encoded_value := oct2unichar(bit2oct(v_str), string_encoding);
    }
  return v_result;
}
```

The decvalue_unichar function first converts the universal charstring value passed in the encoded_value parameter into an octetstring using the unichar2oct function. The string_encoding parameter controls how the characters are converted into octets (in fact how the bitstring sent to the codec contains the characters). The octetstring is subsequently converted into a bitstring by the oct2bit function. This bitstring is then passed as a parameter to the standard decvalue function that performs the actual decoding. In case of successful decoding, the undecoded part of the message is automatically converted from bitstring to octetstring by the bit2oct function and then to universal charstring using the oct2unichar function. This universal charstring is then assigned to the encoded_value parameter. The result of decoding is then returned to the TE, finishing the decvalue_unichar call.

The restrictions in clause 16.1.2 apply.

C.5.5 The encoding to octetstring function

```
encvalue_o(in template (value) any_type inpar,
           in universal charstring encoding_info := "") return octetstring
```

The **encvalue_o** function encodes a value or template into an octetstring. 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 octetstring represents the encoded value of `inpar`, however, the TTCN-3 test system need not make any check on its correctness. The optional `encoding_info` parameter is used for passing additional encoding information to the codec and, if it is omitted, no additional information is sent to the codec.

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.5.6 The decoding from octetstring function

```
decvalue_o(inout octetstring encoded_value,
           out any_type decoded_value,
           in universal charstring decoding_info := "") return integer
```

The **decvalue_o** function decodes an octetstring into a value. The test system shall suppose that the octetstring `encoded_value` represents an encoded instance of the actual type of `decoded_value`. The optional `decoding_info` parameter is used for passing additional decoding information to the codec and, if it is omitted, no additional information is sent to the codec.

If the decoding was successful, then the used octets 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. This value is also returned if the `encoded_value` parameter contains an unitialized value.
- The return value 2 indicates that decoding could not be completed as `encoded_value` did not contain enough octets.

The restrictions in clause 16.1.2 apply.

C.5.7 Retrieving the type of string encoding

```
get_stringencoding(in octetstring encoded_value) return charstring
```

The **get_stringencoding** function analyses the encoded_value and returns the UCS encoding scheme according to clause 10 of ISO/IEC 10646 [2] (see also clause 27.5 of the present document). The identified encoding scheme, or the value "<unknown>", if the type of encoding cannot be determined unanimously, shall be returned as a character string.

NOTE: The initial octet sequence (also known as byte order mark, BOM), when present, allows identifying the encoding scheme unanimously. When it is not present, other symptoms may be used to identify the encoding scheme unanimously; for example, only UTF-8 may have odd number of octets **and** bit distribution according to table 2 of clause 9.1 of ISO/IEC 10646 [2].

EXAMPLE:

```
match ( get_stringencoding('6869C3BA7A'O,charstring:"UTF-8") ) // true
//(the octetstring contains the UTF-8 encoding of the character sequence "hiúz")
```

C.5.8 Removing BOMs of UCS encoding schemes

```
remove_bom(in octetstring encoded_value) return octetstring
```

The **remove_bom** function removes the optional FEFF ZERO WIDTH NO-BREAK SPACE sequence that may be present at the beginning of a stream of serialized (encoded) universal character strings to indicate the order of the octets within the encoding form, as defined in clause 10 of ISO/IEC 10646 [2]. If no FEFF ZERO WIDTH NO-BREAK SPACE sequence present in the `encoded_value` parameter, the function shall return the value of the parameter without change.

Table C.2: Overview of initial octet sequences used for BOM

Coding scheme	initial octet sequence	comments
UTF-8	EF BB BF	signature not required / no effect
UTF-16BE	FE FF	no signature meaning
UTF-16LE	FF FE	no signature meaning
UTF-16	FE FF FF FE	signature (default FE FF)
UTF-32BE	00 00 FE FF	no signature meaning
UTF-32LE	FF FE 00 00	no signature meaning
UTF-32	00 00 FE FF FF FE 00 00	signature (default 00 00 FE FF)

EXAMPLE:

```
remove_bom('FEFF0068006900FA007A'O) // returns '0068006900FA007A'O

remove_bom('BC'O) ) // returns 'BC'O
// note that this octetstring doesn't contain valid UCS character

//example use: automatic decoding of encoded character strings:
oct2unichar(remove_bom(v_myEncodedCharacterSequence),
            get_stringencoding(v_myEncodedCharacterSequence))
```

C.6 Other functions

C.6.1 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 per test component and for the control part 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 in a test component or the control part.

Each time the **rnd** function is initialized with the same seed value, it shall repeat the same sequence of random numbers.

NOTE: For the purpose of keeping parallel testing deterministic, each test component, as well as the control part has its own random seed. This allows for better reproduciblity of test executions. Thus, the rnd function will always use the seed of the component or control part which calls it.

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.6.2 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 TC_MyTestCase1 () runs on MTC system TSI
    {
        var charstring v_tcName := testcasename ();
          // will return the charstring "TC_MyTestCase1"
        :
    }
    :
    testcase TC_MyTestCase2 () runs on MTC system TSI
    {
        :
    }
    :
}
module MyTSModule {
    :
    function f_myStartAPTC() runs on PTC {
        var charstring v_tcName := testcasename ();
          // will return charstring "TC_MyTestCase1", if the function is
          // called by a test component during the execution of TC_MyTestCase1
          // will return charstring "TC_MyTestCase2", if the function is
          // called by a test component when TC_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.6.3 The hostId function

```
hostid(in charstring idkind := "Ipv4orIPv6") return charstring
```

The **hostid** function shall return the host id of the test component or module control executing the **hostid** function in form of a character string. The **in** parameter `idkind` allows to specify the expected id format to be returned.

Predefined `idkind` values are:

- `"Ipv4orIPv6"`: The contents of the returned character string is an Ipv4 address. If no Ipv4 address, but an Ipv6 address is available, a character string representation of the Ipv6 address is returned.
- `"Ipv4"`: The contents of the returned character string shall be an Ipv4 address.
- `"Ipv6"`: The contents of the returned characterstring shall be an Ipv6 address.

The **hostid** function shall return the empty string, if it cannot retrieve any host id or a host id of a kind different from the kind defined by the actual idkind parameter.

The general error causes in clause 16.1.2 apply.

EXAMPLE:

```
// assume

    testcase TC_MyTestCase () runs on MTC system TSI
    {
        :
        var charstring v_myHostId := hostid ("Ipv4");
        :
    }


// assume further the following statement in module control

    execute(TC_MyTestCase(), -, "127.0.0.1");


// In this setting, v_myHostId will have the value "127.0.0.1" after the execution of hostid
```

Annex D (normative): Preprocessing macros

D.0 General

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 c_myConst:= __FILE__;
//c_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 c_myConst:= __BFILE__;
// c_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 c_myConst := __SCOPE__;        // c_myConst contains "MyModule"
    template charstring m_myTemplate := __SCOPE__;  // m_myTemplate contains "m_myTemplate"

type record MyRecord1
{
    charstring field11,
    charstring field12
}

template MyRecord1 m_myTemplate1 (charstring p_p := __SCOPE__) :=
{
    field11 := p_p,
    field12 := __SCOPE__                            // field12 contains "m_myTemplate1"
}

function f_myFunction() {
  var template MyRecord1 v_myvar1 := m_myTemplate1;
    // field11 of m_myTemplate1 will contain the default value of parameter p_p,
    // i.e. "m_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 m_myTemplate2 := {
    field21 := "a",
    field22 := "MyRecord2"                          // a valid specific value matching
}
```

```
template MyRecord2 m_myTemplate3 := {
    field21 := "a",
    field22 := __SCOPE__
    // Causes an error as __SCOPE__ is replaced with "m_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 m_myTemplate4 :=
{
    field31 := "a",
    field32 :=
    {
        field321 := "MyRecord3"                    // a valid specific value matching
    }
}

template MyRecord3 m_myTemplate5 :=
{
    field31 := "a",
    field32 :=
    {
        field321 := __SCOPE__
        // Causes and error as __SCOPE__ is replaced with "m_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 UTF-16 coded representation form (see clause 9.2 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 "UTF-16" };
```

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 Recommendation ITU-T 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.2.2.4 Status values for TTCN-3 objects

Type and constants defined in this clause support the secure usage of the checkstate port operation defined in clause 22.5.5.

The type definition for this type is:

```
type charstring objState    ("Started", "Halted", "Stopped", "Connected", "Mapped", "Linked");
```

Useful constant definitions for working with object states are:

```
const objState STARTED := "Started";
const objState HALTED := "Halted";
const objState STOPPED := "Stopped";
const objState CONNECTED := "Connected";
const objState MAPPED := "Mapped";
const objState LINKED := "Linked";
```

E.2.2.5 Template kinds of TTCN-3 objects

Type and constants defined in this clause support the secure usage of the predefined `istemplatekind` function, described in clause C.3.5.

The type definition for this type is:

```
type charstring TemplateKind ("value", "list", "complement", "AnyValue", "?", "AnyValueOrNone",
"*", "range", "subset", "superset", "omit", "decmatch", "AnyElement", "AnyElementsOrNone",
"permutation", "length", "ifpresent", "pattern");
```

Useful constant definitions for working with template kinds are:

```
const TemplateKind VALUE := "value";
const TemplateKind LIST := "list";
const TemplateKind COMPLEMENT := "complement";
const TemplateKind ANY_VALUE := "AnyValue";
const TemplateKind ANY_VALUE_OR_NONE := "AnyValueOrNone";
const TemplateKind RANGE := "range";
const TemplateKind SUBSET := "subset";
const TemplateKind SUPERSET := "superset";
const TemplateKind OMIT := "omit";
const TemplateKind DECMATCH := "decmatch";
const TemplateKind ANY_ELEMENT := "AnyElement";
const TemplateKind ANY_ELEMENTS_OR_NONE := "AnyElementsOrNone";
const TemplateKind PERMUTATION := "permutation";
const TemplateKind LENGTH := "length";
const TemplateKind IFPRESENT := "ifpresent";
const TemplateKind PATTERN := "pattern";
```

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 Recommendation ITU-T T.50 character type

A type whose distinguished values are single characters of the version of Recommendation ITU-T T.50 [4] complying to the International Reference Version (IRV) as specified in clause 8.2 of Recommendation ITU-T 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

F.0 General

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:

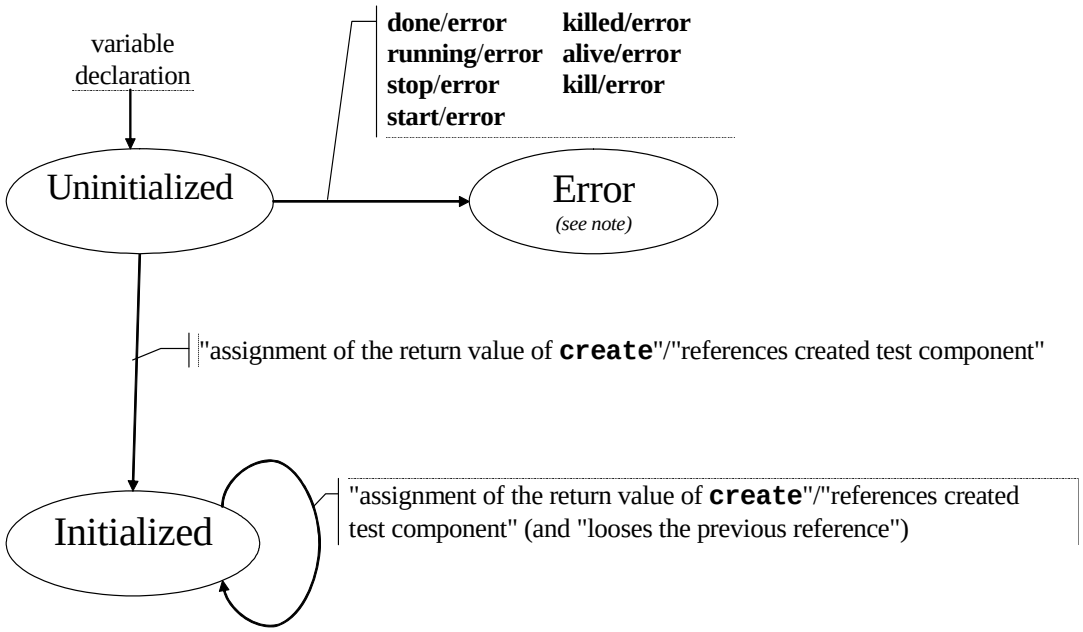
- the states being named and identified as nodes;
- the initial state being identified by an incoming arrow;
- transitions between states connecting two states (not necessarily different states) and identified as arrows;
- transitions being marked with the enabling condition for that transition (i.e. 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 variable of component type. The behaviour related to variables of component type themselves is shown in figure F.1.

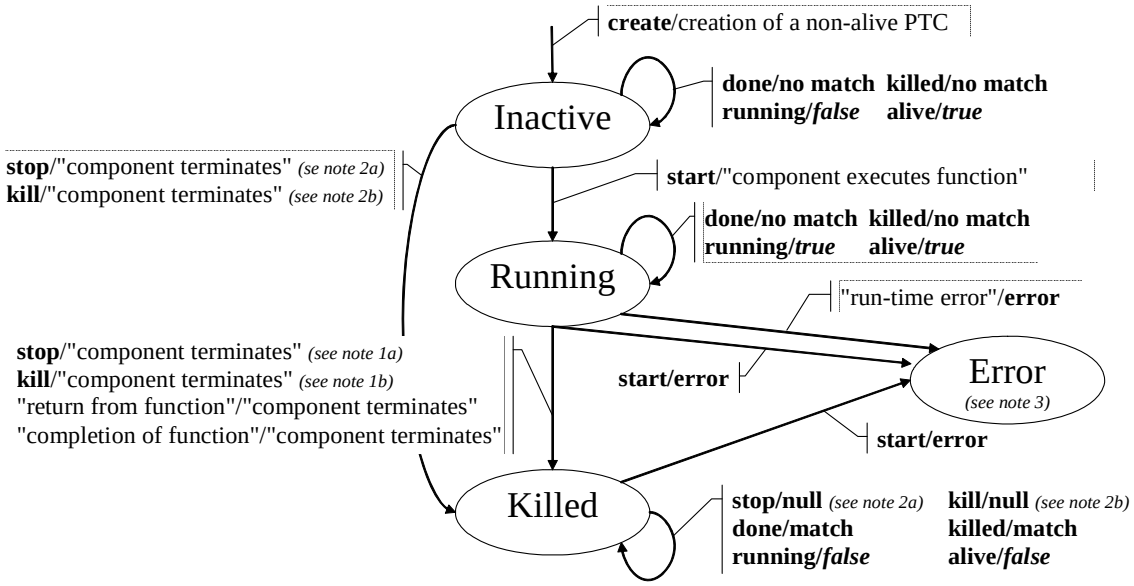


NOTE: Whenever a test component enters its error state, the error verdict is assigned to its local verdict, the test case terminates and the overall test case result will be error.

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.



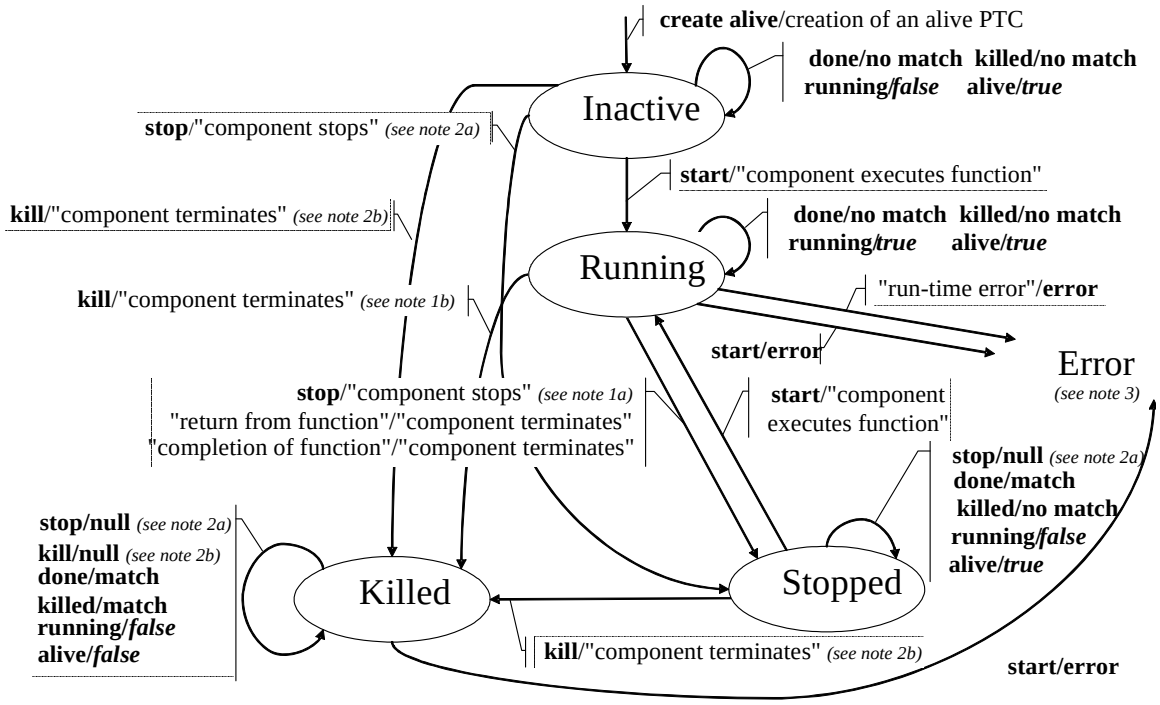
NOTE 1: (a) Stop can be either a stop, self.stop or a stop from another test component.
(b) Kill can be either a kill, self.kill, a kill from another test component or a kill from the test system (in error cases).

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).

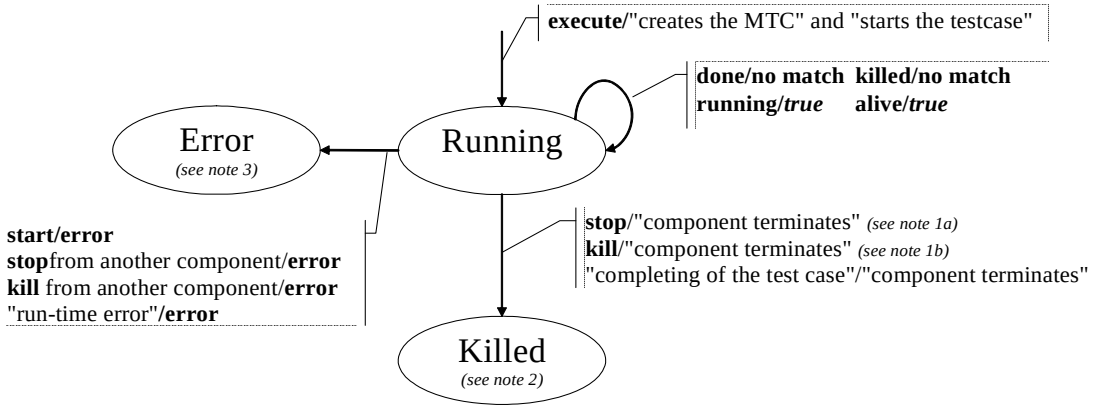
NOTE 2: (a) Stop can be from another test component only. (b) Kill can be from another test component or from the test system (in error cases) only.

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.

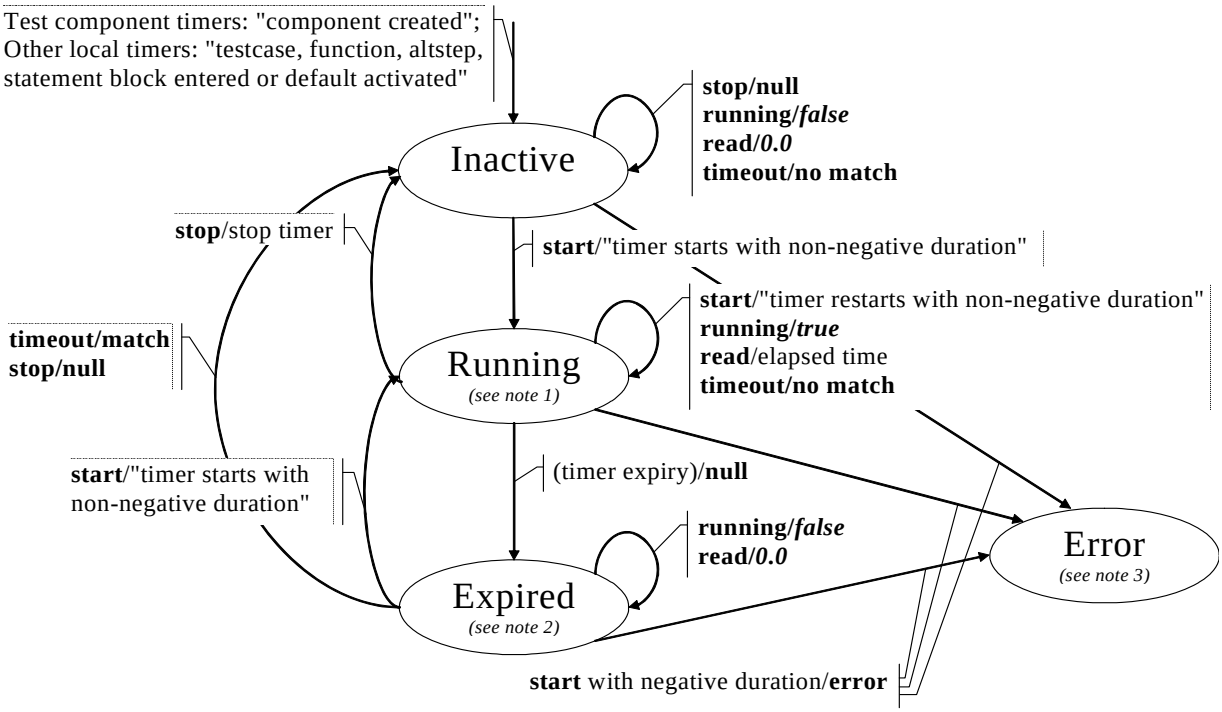


NOTE 1: (a) Stop can be either a stop, self.stop, a stop from another test component.
(b) Kill can be either a kill, self.kill, a kill from another test component or a kill from the test system (in error cases).
NOTE 2: All remaining PTCs are to be killed as well and the testcase terminates.
NOTE 3: 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

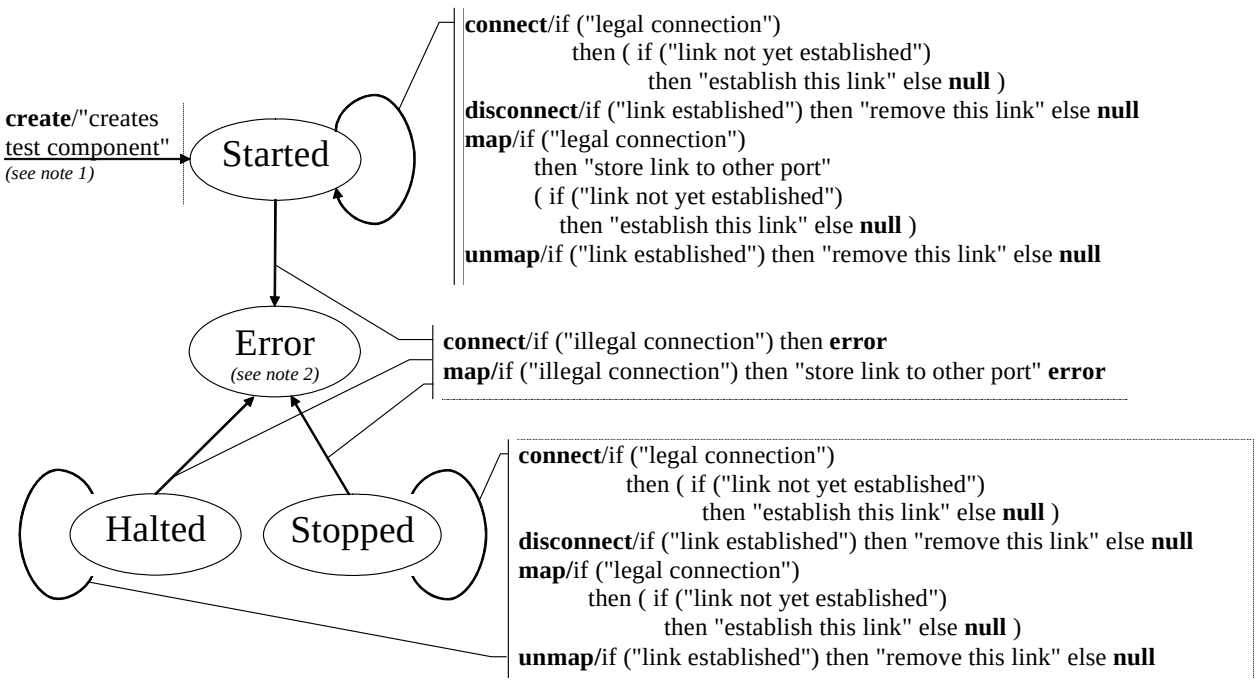
F.3 Ports

F.3.0 General

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.



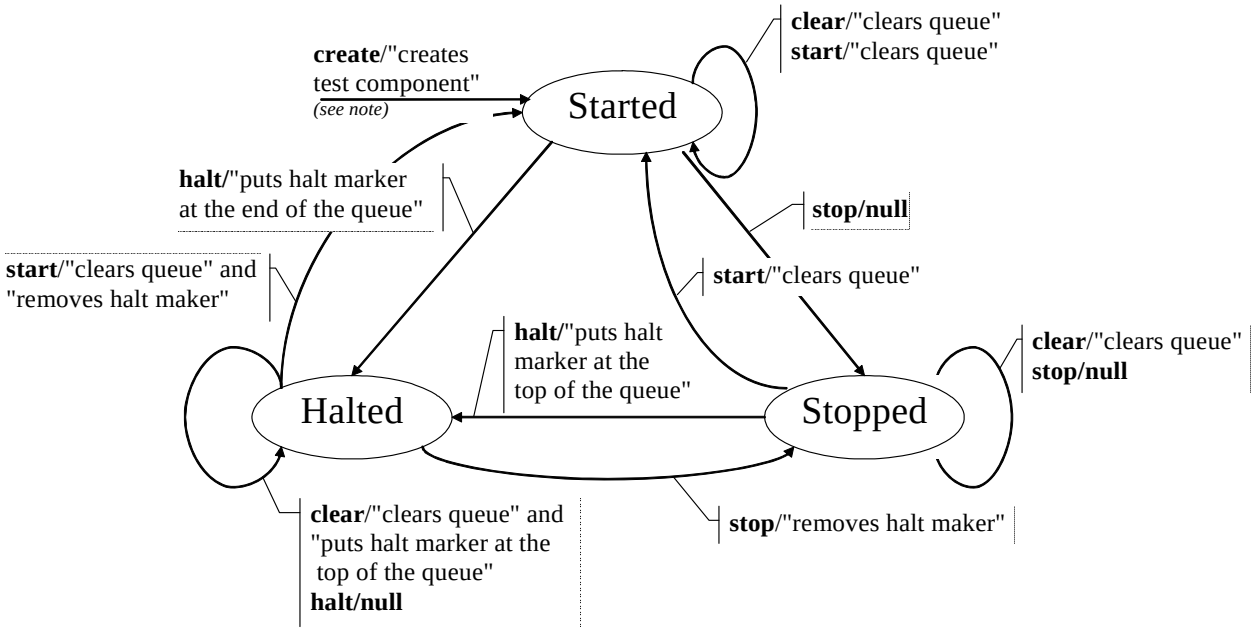
NOTE 1: When creating a PTC the ports of that PTC are created and started; when creating the MTC the ports of the MTC and the ports of the TSI are created and started.
NOTE 2: Whenever a port enters its error state, the test component it belongs to enters also its error state, assigns a local error verdict, 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.

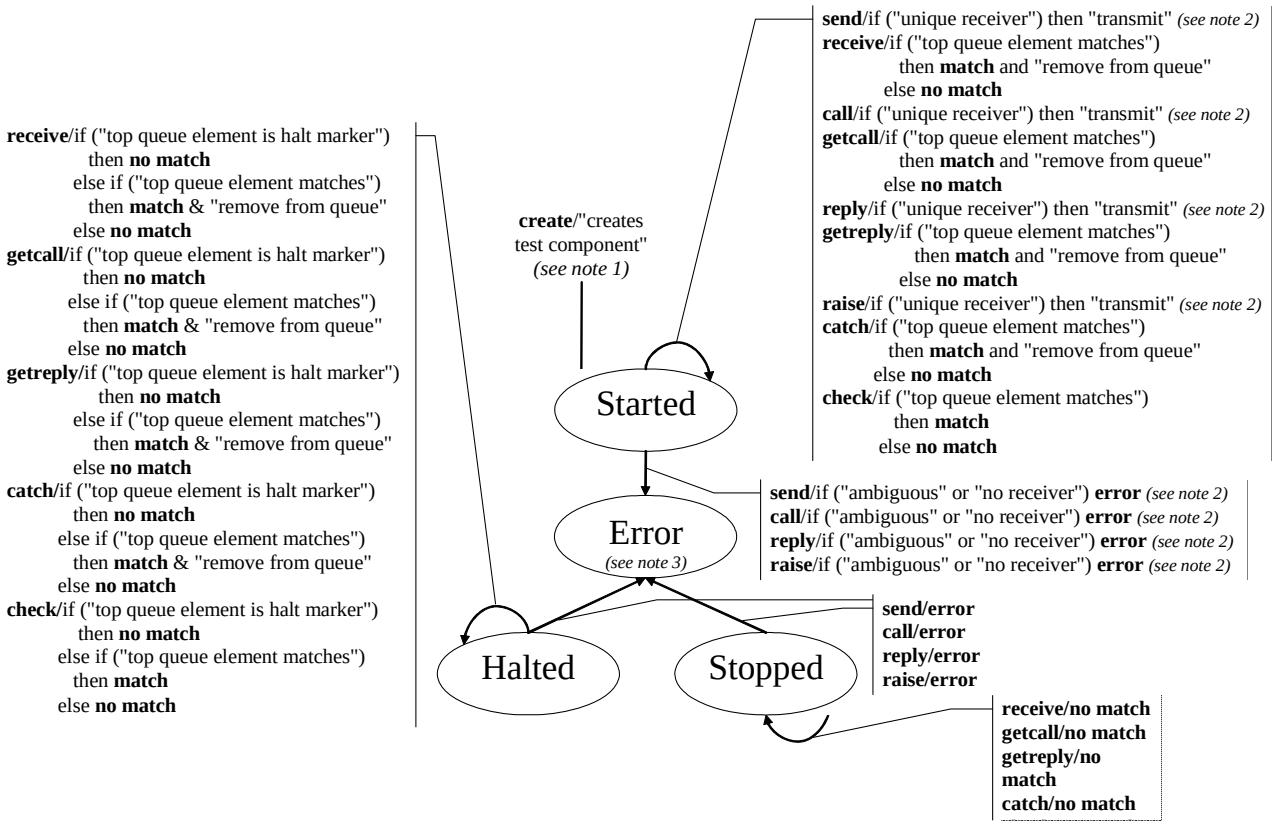


NOTE: When creating a PTC the ports of that PTC are created and started; when creating the MTC the ports of the MTC and the ports of the TSI are created and started.

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 present document.

EXAMPLE (superfluous syntax):

```
module MyModuleWithParameters
{
    modulepar { integer PX_Par0, PX_Par1 := 0;
                boolean PX_Par2 := true
              };
    modulepar { hexstring PX_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 present document.

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 present document.

G.4 sizeof for length of lists

Previous versions of the present document (up to and including V3.2.2) allowed to use the built-in 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 version.

G.5 sizeoftype predefined function

The previous version of the present document (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 version.

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 present document.

G.7 External constants

Previous versions of the present document (up to and including V3.4.1) allowed to use **`external const`**ants. This feature is deprecated and may be fully removed in a future edition of the present document.

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 enumerated values and global names defined in the importing or in another TTCN-3 module. Some tool implementations resolved this issue by allowing prefixing enumerated 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.

G.9 Record of/arrays not compatible to record; set of not compatible with set

Previous versions of the present document (up to and including V4.3.1) did define special cases when record of types and single-dimension arrays would be compatible with record types. These rules are deprecated.

G.10 The "UCS-2" predefined variant attribute string

Previous versions of the present document (up to and including V4.6.1) declared the "UCS-2 variant attribute string to support the UCS-2 coded representation form of ISO/IEC 10646:2003 [i.15] (see clause 14.1 9.2 of ISO/IEC 10646:2003 [i.15]). The use of this string is deprecated, as it is replaced by the predefined variant attribute string "UTF-16".

Similarly, the "UCS-2" and "UCS-4" values of `string_encoding` and serialization parameters, defined in earlier versions of the present document for the **`oct2unichar`**, **`unichar2oct`**, **`encvalue_unichar`** and **`decvalue_unichar`** predefined functions are deprecated.

G.11 Prefixing identifiers of local definitions with module identifiers

Previous versions of the present document (up to and including V4.6.1) did not exclude the possibility to prefix identifier of definitions without global visibility (e.g. templates defined in functions or test cases) with the local module identifier. Prefixing identifiers of local definitions with module identifiers is deprecated and may be fully removed in a future edition of the present document.

G.12 Matching expressions of incompatible types

Previous versions of the present document (up to and including V4.8.1) allowed to use operands of incompatible types in the **`match`** operation, yielding **`false`** as the result. Using an expression and template instance of incompatible types in the **`match`** operation is deprecated and may be fully removed in a future edition of the present document.

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.
- ETSI ES 201 873-1 (V4.3.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language", 2011.
- ETSI ES 201 873-1 (V4.4.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language", 2012.
- ETSI ES 201 873-1 (V4.5.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language", 2013.
- ETSI ES 201 873-1 (V4.6.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language", 2014.
- ETSI ES 201 873-1 (V4.7.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language", 2014.

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
V4.3.1	June 2011	Publication
V4.4.1	April 2012	Publication
V4.5.1	April 2013	Publication
V4.6.1	June 2014	Publication
V4.7.1	June 2015	Publication
V4.8.1	July 2016	Publication
V4.9.1	March 2017	Membership Approval Procedure MV 20170505: 2017-03-06 to 2017-05-05
V4.9.1	May 2017	Publication