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

**ETSI Standard**

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

***Important notice***

The present document can be downloaded from:
<http://www.etsi.org/standards-search>

The present document may be made available in electronic versions and/or in print. The content of any electronic and/or print versions of the present document shall not be modified without the prior written authorization of ETSI. In case of any existing or perceived difference in contents between such versions and/or in print, the only prevailing document is the print of the Portable Document Format (PDF) version kept on a specific network drive within ETSI Secretariat.

Users of the present document should be aware that the document may be subject to revision or change of status. Information on the current status of this and other ETSI documents is available at <https://portal.etsi.org/TB/ETSIDeliverableStatus.aspx>

If you find errors in the present document, please send your comment to one of the following services:
<https://portal.etsi.org/People/CommiteeSupportStaff.aspx>

***Copyright Notification***

No part may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm except as authorized by written permission of ETSI.
The content of the PDF version shall not be modified without the written authorization of ETSI.
The copyright and the foregoing restriction extend to reproduction in all media.

© European Telecommunications Standards Institute 2017.

All rights reserved.

**DECT**TM, **PLUGTESTS**TM, **UMTS**TM and the ETSI logo are Trade Marks of ETSI registered for the benefit of its Members.
**3GPP**TM and **LTE**™ are Trade Marks of ETSI registered for the benefit of its Members and
of the 3GPP Organizational Partners.
**oneM2M** logo is protected for the benefit of its Members
**GSM**® and the GSM logo are Trade Marks registered and owned by the GSM Association.

# 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].

**object reference:** is special kind of value used for instances of component, default, port and timer types which represents a reference to an existing entity in the TE. When used in assignments or parameter passing, only the reference to the entity is copied, but not the entity itself.

**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

NOTE: When passing object references by value, a new reference is created, but the referenced entity remains the same.

**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

**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.

#### 5.4.1.1 Formal parameters of kind value

Values of all basic types, all user-defined types, address type, component type, port type, timers 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 formal parameters of port, timer or default type 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.

l) 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***

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 Void

#### 5.4.1.4 Void

### 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 of any TTCN‑3 type. 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 }

### 6.2.6 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, default and timer 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.

### 6.2.8 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.

Values of the **default** type are object references and follow specific rules for this kind of values.

### 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* [ "," ] }+ ";") }

"}"

TTCN-3 allows to define constants, variables and parameters of a port type. These constants, variables or parameters can contain a reference to an existing component port or a special value **null**. The special value **null** represents an unspecific port reference, e.g. can be used for the initialization of variables of a port type.

Port type values are object references and follow specific rules for this kind of values.

***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.

Component instances are object references and follow specific rules for this kind of values.

***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 systeminterface 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());

 }

 :

 }

### 6.2.14 The timer type

TTCN‑3 allows to define timer constants, variables and parameters. These constants, variables or parameters can contain a reference to an existing timer or a special value **null**. The special value **null** represents an unspecific timer reference, e.g. can be used for the initialization of variables of a timer type.

Timer references have meaning only within the test component instances where the timer is defined, i.e. a timer reference assigned to a timer variable in a test component instance "a1" of type "A" has no meaning in a test component instance "a2" of type "A".

The values of timer type are object references and follow specific rules for this kind of values.

### 6.3.6 Type compatibility of port types

For variables, constants and parameters of port types, the reference to a port "b" of type "B" is compatible to type "A" if type "B" and type "A" are equal or synonym types.

### 6.3.7 Type compatibility of timer types

For variables, constants and parameters of timer types, the reference to a timer is compatible with any other timer reference.

### 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, default, timer or component type and shall not be of a structured type that contains a sub-element of port or timer 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");

 ...

 }

# 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) void

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.

NOTE: The only value that can be assigned to global constants of default, component, port or timer type or component constants of default or component types is the special value **null**.

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**;

### 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, two port, two timer or two component values are equal if and only if they contain a refererence to the same entity (i.e. they designate the same default, port, timer 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

# 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, default, port or timer 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;

 **var** **timer** v\_timer1;

 **timer** t\_myTimer1;

 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

 v\_timer1 := t\_myTimer1;

## 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.

A timer declared and started in scope units such as functions cease to exist when the scope unit is left unless there’s a  constant, variable or parameter defined in the current or higher scope unit or in an activated altstep that contains a reference to it. In this case, the timer is kept as long as at least one constant, variable or parameter of the current or higher scope unit or an activated altstep contain a reference to it.

NOTE 1: Timers that ceased to exist do not contribute to the test behaviour.

***Restrictions***

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

1. 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).
2. 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 := 5**E-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.

# 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:

1. they are a way to organize and to re-use test data, including a simple form of inheritance;
2. they can be parameterized;
3. they allow matching mechanisms;
4. 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:

1. Templates shall not be of **default**, port or timer type.
2. Templates shall not be of a structured type that contains fields of **default**, port or timer type on any level of nesting.

NOTE 2: The **anytype** type does not include the **default** type, port and timer types (see clause 6.2.6), so that restriction b) does not apply to anytype templates.

1. The expression or template body initializing a template shall evaluate to a value or template, which is type compatible with the template being declared.
2. 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.
3. 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.
4. 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

## 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:

1. 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
2. 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.

c) All formal parameter types of the test case shall neither be of a port, default or timer type nor should contain a direct or indirect element or field of a port, default or timer type.

***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

 }

## 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 use the rules of passing by value. If the variable being assigned is of a type whose values are object references, only the reference is copied, but the referenced object (e.g. component, timer or port) is not. In all other cases, the content being assigned shall be a copy of the evaluated right‑hand side.

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:

1. The right‑hand side of an assignment shall evaluate to a value, template, port or timer, which is type compatible with the variable at the left-hand side of the assignment.
2. 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.
3. The right‑hand side of an assignment shall evaluate to an object that is at least partially initialized.
4. 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.
5. 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, v\_myList2, v\_myList3;

 v\_myList1 := {"value1", "value2", "value3" }; // v\_myList1 is completely initialized

 v\_myList2 := v\_myList1; // v\_myList2 is equal to {"value1", "value2",
 // "value3" }

 v\_myList2.field1 := "missing"; // only v\_myList2 value changes to
 // {"missing", "value2", "value3" };
 // v\_myList1 still contains {"value1", "value2",
 // "value3" } after the assignment

 v\_myList3.field2 := "newvalue"; // v\_myList3 is partilly initialized

 // field1 and field3 remain uninitialized

 v\_myList1 := v\_myList3; // v\_myList1 become partially initialized,

 // field2 has the value "newvalue"

 // field1 and field3 are uninitialized

EXAMPLE 4:

 **var** **timer** v\_timer1;

 **timer** t\_timer1 := 100.0;

 t\_timer1.**start**;

 v\_timer1 := t\_timer1;

 v\_timer1.**stop**; // stopping the timer using the variable reference

 log(t\_timer.**running**); // logs false as the previous statement stopped the original timer

### 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:

1. void
2. void
3. An altstep that is activated as a default shall only have **in** 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.

 :

 }

### 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:

1. 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.
2. 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).
3. 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 a port, default or timer type nor should contain a direct or indirect element or field of a port, default or timer 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<>

## 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***

*Expression* "." **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.

g) The *Expression* preceding the **send** keyword shall be of a port type.

***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***

(*Expression* | **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 PortArrayRefshallbe areference 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.

n) The *Expression* preceding the **receive** keyword shall be of a port type.

***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***

( *Expression* | **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 PortArrayRefshallbe areference 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.

k) The *Expression* preceding the **trigger** keyword shall be of a port type.

***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***

*Expression* "." **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.

n) The *Expression* preceding the **call** keyword shall be of a port type.

***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***

( *Expression* | **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 PortArrayRefshallbe areference 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.

n) The *Expression* preceding the **getcall** keyword shall be of a port type.

***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***

*Expression* "." **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.

g) The *Expression* preceding the **receive** keyword shall be of a port type.

***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***

( *Expression* | **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 PortArrayRefshallbe areference 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.

n) The *Expression* preceding the **getreply** keyword shall be of a port type.

***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***

*Expression* "." **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).

g) The *Expression* preceding the **raise** keyword shall be of a port type.

***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***

( *Expression* | **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 PortArrayRefshallbe areference 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:

1. 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.
2. 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.
3. 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.
4. Exception values accepted by *catch any exception* shall not be assigned to a variable, i.e. the **value** clause shall not be present.
5. 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.
6. 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.
7. The *PortArrayRef* shall be a reference to a completely initialized port array.
8. The index redirection shall only be used when the operation is used on an any from port array construct.
9. 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.
10. 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.
11. 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.
12. 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.

1. 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.
2. The *Expression* preceding the **catch** keyword shall be of a port type.

***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***

( *Expression* | **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 PortArrayRefshallbe areference 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:

1. 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.
2. 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.
3. The *PortArrayRef* shall be a reference to a completely initialized port array.
4. The index redirection shall only be used when the operation is used on an any from port array construct.
5. 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.
6. 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.
7. 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.
8. 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.
9. n) The *Expression* preceding the **check** keyword shall be of a port type.

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***

( *Expression* | ( **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***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

1. The *Expression* preceding the **clear** keyword shall be of a port type.

***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***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

1. The *Expression* preceding the **start** keyword shall be of a port type.

***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***

( *Expression* | ( **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***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

1. The *Expression* preceding the **stop** keyword shall be of a port type.

***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***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

1. The *Expression* preceding the **halt** keyword shall be of a port type.

***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***

( *Expression* | ( **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.

1. Calling the **checkstate** operation with a **charstring** parameter not listed in a) shall lead to an error.
2. The *Expression* preceding the **checkstate** keyword shall be of a port type.

***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***

*Expression*

"." **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:

1. 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).
2. The *Expression* preceding the **start** keyword shall be of the timer type.

***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***

( *Expression* |

 **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***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

1. The *Expression* preceding the **stop** keyword shall be of the timer type.

***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***

*Expression*

"." **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***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

1. The *Expression* preceding the **read** keyword shall be of the timer type.

***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***

( *Expression* |

 **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** TimerArrayRefnotation 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:

1. *TimerArrayRef* shall be a reference to a completely initialized timer array.
2. The index redirection shall only be used for **any from** timer array running operations.
3. 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.
4. 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.
5. The *Expression* preceding the **running** keyword shall be of the timer type.

***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***

( *Expression* |

 **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** TimerArrayRefnotation 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:

1. The **timeout** operation does not return any value and therefore shall not be used in an expression.
2. *TimerArrayRef* shall be a reference to a completely initialized timer array.
3. The index redirection shall only be used for **any from** timer array timeout operations.
4. 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.
5. 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.
6. The *Expression* preceding the **timeout** keyword shall be of the timer type.

***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** |

### A.1.6.5 Type

Type ::= [PredefinedType](#TPredefinedType) | [ReferencedType](#TReferencedType)

PredefinedType ::= [BitStringKeyword](#TBitStringKeyword) |

 [BooleanKeyword](#TBooleanKeyword) |

 [CharStringKeyword](#TCharStringKeyword) |

 [UniversalCharString](#TUniversalCharString) |

 [IntegerKeyword](#TIntegerKeyword) |

 [OctetStringKeyword](#TOctetStringKeyword) |

 [HexStringKeyword](#THexStringKeyword) |

 [VerdictTypeKeyword](#TVerdictTypeKeyword) |

 [FloatKeyword](#TFloatKeyword) |

 [AddressKeyword](#TAddressKeyword) |

 [DefaultKeyword](#TDefaultKeyword) |

 [AnyTypeKeyword](#TAnyTypeKeyword) |

 TimerKeyword

BitStringKeyword ::= "bitstring"

BooleanKeyword ::= "boolean"

IntegerKeyword ::= "integer"

OctetStringKeyword ::= "octetstring"

HexStringKeyword ::= "hexstring"

VerdictTypeKeyword ::= "verdicttype"

FloatKeyword ::= "float"

AddressKeyword ::= "address"

DefaultKeyword ::= "default"

AnyTypeKeyword ::= "anytype"

CharStringKeyword ::= "charstring"

UniversalCharString ::= [UniversalKeyword](#TUniversalKeyword) [CharStringKeyword](#TCharStringKeyword)

UniversalKeyword ::= "universal"

ReferencedType ::= [ExtendedIdentifier](#TExtendedIdentifier) [[ExtendedFieldReference](#TExtendedFieldReference)]

TypeReference ::= [ExtendedIdentifier](#TExtendedIdentifier)

ArrayDef ::= {"[" [SingleExpression](#TSingleExpression) [".." [SingleExpression](#TSingleExpression)] "]"}+

/\* STATIC SEMANTICS - ArrayBounds will resolve to a non negative value of integer type \*/