ETSI ES 201 873-1 V4.12.1 (2020-05)

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-201873-1V4.12.1

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 prevailing version of an ETSI deliverable is the one made publicly available in PDF format at [www.etsi.org/deliver](http://www.etsi.org/deliver).

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.

© ETSI 2020.

All rights reserved.

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

# 3 Definition of terms, symbols and abbreviations

## 3.1 Terms

For the purposes of the present document, the terms 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 elements 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:** dynamic test behaviour, which is either a testcase, a function, or an altstep definition

**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 6.2.15 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

NOTE 1: 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 fulfils the requirement of being "at least partially initialized".

NOTE 2: 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 data types:** collection of all data types, component types and structured types whose sub-elements are component data types

**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

**control behaviour:** collection of module control functions with the name **control** and functions and altsteps called by **control** directly or through other control functions or altsteps, and are used for the dynamic execution of test cases

NOTE: Such functions and altsteps are called control functions and control altsteps respectively. Module control functions can be used as an entry point of executing a test suite. Declaring functions or altsteps with the modifier **@control** explicitly allows to distinguish them from test case behaviour definitions in their special role. Module control functions and behaviour definitions with the **@control** modifier are called explicit control behaviour definitions, i.e explicit control functions and explicit control altsteps.

**data types:** all types whose values or sub-elements cannot contain object references

NOTE: Data types include simple basic types, basic string types, and the special data type anytype. Data types also include all structured types where all their sub-elements are of a data type. All user defined types based on a data type are data types as well. See more details in 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:** value or template instance that is declared to be fuzzy and consequently the expression, initializing or partly initializing it (including actual parameters passed to **in** formal parameters), is subject to lazy evaluation

NOTE: 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

**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:** evaluation of an expression, 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 another place than the left hand side of an assignment or an actual parameter passed to a fuzzy or lazy formal parameter

NOTE: 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:** value or template instance for which the expression, initializing or partly initializing it (including actual parameters passed to in formal parameters), is subject to lazy evaluation

NOTE: 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 Control component (MCC):** The main control component is the component that is started when executing a testsuite by running a control function. The control function is the behaviour being executed by the MCC.

**main test component (MTC):** See Recommendation ITU‑T X.292 [3].

**object:** instance of one of the object types (component, default, port and timer)

NOTE: Objects of type default, port or timer, which are owned by the component that instantiated them, are local objects while objects of type component are global objects. Global objects can be referenced from other component scopes while references to local objects can only be used by the component they are bound to.

**object reference:** 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

NOTE: When used in assignments or parameter passing, only the reference to the entity is copied, but not the entity itself. An object reference can also be initialized with the special value null in which case it does not reference an object.

**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 parameter has been initialized during the invocation and 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: Formal an **out** parameters are uninitialized (unbound) when the invoked object is entered.

**parallel Control component (PCC):** A parallel control component is a component created inside control behaviour, but not inside testcase behaviour. PCCs can be created either by the main control component or by other PCCs.

**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**

NOTE: 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 behaviour).

**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 functions

**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.

# 21 Configuration Operations

## 21.0 General

Configuration operations are used to set up and control test components and their connections as well as control components and their connections. They are summarized in table 20.

***Restrictions***

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

1. These operations shall only be used in:
* TTCN‑3 test cases;
* behaviour invoked directly or indirectly from a test case or from behaviour started on a ptc.
1. They shall not be present in:
* control behaviour;
* declarations inside component type definitions; or
* functions invoked directly or indirectly from declarations inside component type definitions.

Table 20: Overview of TTCN‑3 configuration operations

| Operation | Explanation | Syntax Examples |
| --- | --- | --- |
| **Connection Operations** |
| connect | Connects the port of one test component to the port of another test component | **connect**(ptc1:p1, ptc2:p2); |
| disconnect | Disconnects two or more connected ports | **disconnect**(ptc1:p1, ptc2:p2); |
| map | Maps the port of one test component to the port of the test system interface | **map**(ptc1:q, **system**:sutPort1); |
| unmap | Unmaps two or more mapped ports | **unmap**(ptc1:q, **system**:sutPort1); |
| **Component Operations** |
| create | Creation of a normal or alive component, the distinction between normal and alive components is made during creation(MTC behaves as a normal test component) | Non-alive components:**var** PTCType c := PTCType.**create**;Alive components:**var** PTCType c := PTCType.**create** **alive**; |
| start | Starting behaviour on a component, starting behaviour does not affect the status of component variables, timers or ports | c.**start**(PTCBehaviour()); |
| stop | Stopping behaviour on a component | c.**stop**; |
| kill | Causes a component to cease to exist | c.**kill**; |
| alive | Returns true if the component has been created and is ready to execute or is already executing behaviour; otherwise returns false | **if** (c.**alive**) … |
| running | Returns true as long as the component is executing behaviour; otherwise returns false | if (c.**running**) … |
| done | Checks whether the behaviour running on a component has terminated | c.**done**; |
| killed | Checks whether a component has ceased to exist | c.**killed** { … } |
| **Test Case Operations** |
| stop | Terminates the test case with the test verdict **error** | **testcase**.**stop** ( … ); |
| **Reference Operations** |
| mtc | Gets the reference to the MTC | **connect**(**mtc**:p, ptc:p); |
| system | Gets the reference to the test system interface | **map**(c:p, **system**:sutPort); |
| self | Gets the reference to the test component that executes this operation | **self**.**stop**; |

## 21.1 Connection Operations

### 21.1.0 General

The ports of a component can be connected to ports of other components or to the ports of the test system interface (see figure 10). In the case of connections between two test components or two control components, the **connect** operation shall be used. When connecting a test component or control component port to a test system port the **map** operation shall be used. The **connect** operation directly connects one port to another with the **in** side connected to the **out** side and vice versa. The **map** operation on the other hand can be seen purely as a name translation defining how communications streams can be referenced.



Figure 10: Illustration of the connect and map operations

### 21.1.1 The Connect and Map operations

The **connect** operation is used to setup connections between non-system components. The **map** operation are used to setup connections to the SUT.

***Syntactical Structure***

**connect** "(" *ComponentRef* ":" *Port* "," *ComponentRef* ":" *Port* ")"

**map** "(" *ComponentRef* ":" *Port* "," *ComponentRef* ":" *Port* ")"

[ **param** "(" [ { *ActualPar* [","] }+ ] ")" ]

***Semantic Description***

With both the **connect** operation and the **map** operation, the ports to be connected are identified by the component references of the components to be connected and the names of the ports to be connected.

The operation **mtc** identifies the MTC, the operation **system** identifies the test system interface and the operation **self** identifies the component in which **self** has been called (see clause 6.2.11). All these operations can be used for identifying and connecting ports.

Both the **connect** and **map** operations shall be only invoked from places specified in 21.0.. Before either operation is called, the components to be connected shall have been created and their component references shall be known together with the names of the relevant ports.

Applying a **map** or **connect** operation to ports which are already mapped or connected has no effect on the test behaviour or test configuration, i.e. test execution continues as if the operation has not been invoked.

NOTE 1: Please note that also triMap or tciConnect respectively will not be invoked in such a case.

The **map** operation provides an optional parameter list for configuration purposes. This allows to pass values needed for dynamic runtime configuration. If a parameter list is present, the actual parameters shall conform to the **map** **param** clause of the port type declaration of the system port used.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clauses 5 and 21 and shown in table 16, the following restrictions apply:

a) For both the **connect** and **map** operations, only consistent connections are allowed.

 Assuming the following:

1) ports PORT1 and PORT2 are the ports to be connected or mapped;

2) inlist-PORT1 defines the messages or procedures of the in-direction of PORT1;

3) outlist-PORT1defines the messages or procedures of the out-direction of PORT1;

4) inlist-PORT2 defines the messages or procedures of the in-direction of PORT2; and

5) outlist-PORT2 defines the messages or procedures of the out-direction of PORT2.

b) The **connect** operation is allowed if and only if:

1) outlist-PORT1 ⊆ inlist-PORT2 and outlist-PORT2 ⊆ inlist-PORT1; and

2) neither PORT1 nor PORT2 are system port references; and

3) atleast one of outlist-PORT1 or outlist-PORT2 is not empty.

c) The **map** operation is allowed if and only if:

1) PORT1 is a component port reference *and* PORT2 is a system port reference; and

2) outlist‑PORT1 ⊆ outlist-PORT2 *and* inlist-PORT2 ⊆ inlist-PORT1; and

3) at least one of outlist-PORT1 or inlist-PORT2 is not empty.

NOTE 2: Please note that PORT1 and PORT2 can occur in any order, thus the system adapter port can be either the first or the second operand of the map operation.

d) In all other cases, the operations shall not be allowed.

e) Since TTCN‑3 allows dynamic configurations and addresses, not all of these consistency checks can be made statically at compile-time. All checks, which could not be made at compile-time, shall be made at runtime and shall lead to a test case error when failing.

f) In addition, the restrictions on allowed and disallowed connections described in clause 9.1 apply.

g) In **map** operations, **param** clauses are optional. If in a **map** operation a **param** clause is present, exactly one of the components referenced by the operation shall be the **system** component reference, the type of the system component shall be known in the context of the operation either via a **system** clause or via a **runs on** clause in a **testcase** without **system** clause, the type of the system port to which the operation is applied shall include a **map param** declaration, and the actual parameters shall conform to the **map** **param** clause of the port type declaration of the system port used.

h) If the type of the component referenced in a connection operation is known (either when the component reference is a variable or value returned from a function or the type is defined in the runs on, mtc or system clause of the calling function), the referenced port declaration shall be present in this component type.

***Examples***

EXAMPLE 1: Simple map and connect:

 // It is assumed that the ports Port1, Port2, Port3 and PCO1 are properly defined and declared

 // in the corresponding port type and component type definitions

 :

 **var** MyComponentType v\_myNewPTC;

 v\_myNewPTC := MyComponentType.**create**;

 :

 **connect**(v\_myNewPTC:port1, **mtc**:port3);

 **map**(v\_myNewPTC:port2, **system**:pCO1);

 :

 // In this example a new component of type MyComponentType is created and its reference stored

 // in variable v\_myNewPTC. Afterwards in the connect operation, port1 of this new component

 // is connected with port3 of the MTC. By means of the map operation, port2 of the new component

 // is then connected to port pCO1 of the test system interface

EXAMPLE 2: Parameterized map:

 :

 **var** MyConfigType v\_myConfig := { option := 1, lock := **false**};

 :

 **map**(**mtc**:port4, **system**:pCO2) **param** (v\_myConfig);

 :

 // In this example by means of the map operation, port4 of the MTC is connected to the port pCO2

 // of the test system interface, and additionally a parameter containing configuration options

 // for the connection is passed.

EXAMPLE 3: Port visibility:

 **type** **port** P **message** { **inout** **integer**; }

 **type** **component** C1 { **port** P p1; }

 **type** **component** C2 { **port** P p1, p2; }

 **testcase** TC **runs on** C1 **system** C1

 {

 **var** C1 v\_ptc := C2.**create**; // valid assignment, instance of C2 is compatible with C1 type

 **connect** (**self**:p1, v\_ptc:p1); // valid, p1 is present in C1 type definition

 **disconnect** (**self**:p1, v\_ptc:p1);

 **connect** (**self**:p1, v\_ptc:p2); // invalid, although the real instance in v\_ptc is of the

 // C2 type, the variable itself is of the C1 type making the p2 port invisible to the

 // connection operation
 **connect** (v\_ptc:p1, **system**:p1); // invalid, connect parameters shall not contain

 // a system port reference

 }

### 21.1.2 The Disconnect and Unmap operations

The **disconnect** and **unmap** operations are the opposite operations of **connect** and **map**.

***Syntactical Structure***

**disconnect** [ ( "(" *ComponentRef* ":" *Port* "," *ComponentRef* ":" *Port* ")" ) |

 ( "(" *PortRef* ")" ) |

 ( "(" *ComponentRef* ":" **all** **port** ")" ) |

 ( "(" **all** **component** ":" **all** **port** ")" ) ]

**unmap** [ ( "(" *ComponentRef* ":" *Port* "," *ComponentRef* ":" *Port* ")"

 [ **param** "(" [ { *ActualPar* [","] }+ ] ")" ] ) |

 ( "(" *PortRef* ")" [ **param** "(" [ { *ActualPar* [","] }+ ] ")" ] ) |

 ( "(" *ComponentRef* ":" **all** **port** ")" ) |

 ( "(" **all** **component** ":" **all** **port** ")" ) ]

***Semantic Description***

The **disconnect** and **unmap** operations perform the disconnection (of previously connected) ports of non-system components and the unmapping of (previously mapped) ports of non-system components and ports in the test system interface.

Both, the **disconnect** and **unmap** operations can be called from any component if the relevant component references together with the names of the relevant ports are known. A **disconnect** or **unmap** operation has only an effect if the connection or mapping to be removed has been created beforehand.

To ease **disconnect** and **unmap** operations related to all connections and mappings of a component or a port, it is allowed to use **disconnect** and **unmap** operations with one argument only. This one argument specifies one side of the connections to be disconnected or unmapped. The **all port** keyword can be used to denote all ports of a component.

The usage of a **disconnect** or **unmap** operation without any parameters is a shorthand form for using the operation with the parameter **self**:**all port**. It disconnects or unmaps all ports of the component that calls the operation.

The **all component** keyword shall only be used in combination with the **all port** keyword, i.e. **all** **component**:**all** **port**, and shall only be used by the MTC. Furthermore, the **all component**:**all port** argument shall be used as the one and only argument of a **disconnect** or **unmap** operation and it allows to release all connections and mappings of the test configuration.

Similar to the **map** operation, **unmap** provides an optional parameter list for configuration purposes. If a parameter list is present, the actual parameters shall conform to the **unmap** **param** clause of the port type declaration of the system port used. It allows to pass values needed for dynamic runtime configuration.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clauses 5 and 21 and shown in table 16, the following restrictions apply:

a) In an **unmap** operation, a **param** clause shall only be present if the system port to which the **param** clause belongs to is explicitly referenced.

b) In **unmap** operations, **param** clauses are optional. If in an **unmap** operation a **param** clause is present, exactly one of the components referenced by the operation shall be the **system** component reference, the type of the system component shall be known in the context of the operation either via a **system** clause or via a **runs on** clause in a **testcase** without **system** clause, the type of the system port to which the operation is applied shall include an **unmap param** declaration and the actual parameters shall conform to the **unmap** **param** clause of the port type declaration of the system port used.

c) If the type of the component referenced in a connection operation is known (either when the component reference is a variable or value returned from a function or the type is defined the runs on, mtc or system clause of the calling function), the referenced port declaration shall be present in this component type.

d) The disconnect operation parameters shall not contain a system port reference.

***Examples***

EXAMPLE 1: Disconnect/unmap for specific connections:

 **connect**(myNewComponent:port1, **mtc**:port3);

 **map**(myNewComponent:port2, **system**:pCO1);

 :

 **disconnect**(myNewComponent:port1, **mtc**:port3); // disconnect previously made connection

 **unmap**(myNewComponent:port2, **system**:pCO1); // unmap previously made mapping

EXAMPLE 2: Disconnect/unmap for a component:

 **disconnect**(myNewComponent:port1); // disconnects all connections of Port1, which
 // is owned by component myNewComponent.

 **unmap**(myNewComponent:**all port**); // unmaps all ports of component myNewComponent

EXAMPLE 3: Disconnect/unmap for "self":

 **disconnect**; // is a shorthand form for …

 **disconnect**(**self**:**all** **port**); // which disconnects all ports of the component

 // that called the operation

 :

 **unmap**; // is a shorthand form for …

 **unmap**(**self**:**all** **port**); // which unmaps all ports of the component

 // that called the operation

EXAMPLE 4: Disconnect/unmap for "all component":

 **disconnect**(**all component**:**all** **port**); // the MTC disconnects all ports of all

 // components in the test configuration.

 :

 **unmap**(**all component**:**all** **port**); // the MTC unmaps all ports of all

 // components in the test configuration.

## 21.2 Test case operations

### 21.2.0 General

Test case operations address the entire test case by using the keyword testcase. Currently, the test case stop operation is the only test case operation. It specifies an immediate stop of the test case behaviour with an error verdict.

### 21.2.1 Test case stop operation

The testcase stop operation defines a user defined immediate termination of a test case with the test verdict **error** and an (optional) associated reason for the termination. Such an immediate stop of a test case is required for cases where a user defined behaviour that does not contribute to the test outcome behaves in an unexpected manner which leads to a situation where the continuation of the test case makes no more sense.

***Syntactical Structure***

**testcase** "." **stop** [ "(" { ( *FreeText* | *TemplateInstance* ) [","] } ")" ]

***Semantic Description***

The test case stop operation causes an immediate stop of the entire test case behaviour with the verdict **error**. In addition, the test case stop operation provides the means to specify the reason for the immediate termination of a test case by writing one or more items to some logging device associated with the test control or the test component in which the operation is used. Items to be logged shall be identified by a comma‑separated list in the argument of the test case stop operation. The argument of the test case stop operation shall follow the same restrictions as the argument of the log statement (see clause 19.11).

***Restrictions***

No specific restrictions in addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 16.

***Examples***

 **testcase.stop**("Unexpected Termination");

 // The test case stops the an error verdict and the string "Unexpected Termination"

 // is written to some log device of the test system

## 21.3 Component Operations

### 21.3.0 General

Component operations are used to create, start, stop and kill test or control components. They can also be used to check if components are alive or running, or react to them being done or killed.

### 21.3.1 The Create operation

The **create** operation is used to create test components inside testcase behavior and control components inside control behavior.

***Syntactical Structure***

*ComponentType* "." **create** [ "(" *Expression* ["," *Expression*] ")" ] [ **alive** ]

***Semantic Description***

The MTC is the only test component, which is automatically created when a test case starts. All other test components (the PTCs) shall be created explicitly during test execution by **create** operations. Likewise, the main control component (MCC) is the only control component which is created automatically when starting the main control function. All other parallel control components (the PCCs) are created explicitly during control execution by create operations. A component is created with its full set of ports of which the input queues are empty and with its full set of constants, variables and timers. Furthermore, if a port is defined to be of the type **in** or **inout** it shall be in a listening state ready to receive traffic over the connection.

All component variables and timers are reset to their initial value (if any) and all component constants are reset to their assigned values when the component is explicitly or implicitly created.

Two types of parallel components are distinguished: a component that can execute a behaviour only once and a component that is kept alive after termination of a behaviour and can be therefore reused to execute another behaviour. The latter is created using the additional **alive** keyword. An alive-type component shall be destroyed explicitly using the **kill** operation (see clause 21.3.4), whereas a non-alive component is destroyed implicitly after its behaviour terminates. Termination of a test case, i.e. the MTC, terminates all PTCs that still exist, if any. Termination of the main control behavior terminates all PCCs that still exist.

Since all test components and ports are implicitly destroyed at the termination of each test case, each test case shall completely create its required configuration of components and connections when it is invoked. Likewise, the main control component shall create its required configuration of parallel control components and connections.

The **create** operation shall return the unique component reference of the newly created instance. The unique reference to the component will typically be stored in a variable (see clause 6.2.10.1) and can be used for connecting instances and for communication purposes such as sending and receiving.

Optionally, a name can be associated with the newly created component instance. The test system shall associate the names 'MTC' to the MTC and 'SYSTEM' to the test system interface automatically at creation. Associated component names are not required to be unique.

The component instance name is used for logging purposes (see clause 19.11) only and shall not be used to refer to the component instance (the component reference shall be used for this purpose) and has no effect on matching.

Also optionally, a host id can be associated with the newly created component instance. If a host id is provided, the **create** operation shall cause a test case error, if the component cannot be deployed on the specified host.

Components can be created at any point in a behaviour definition providing full flexibility with regard to dynamic configurations (i.e. any component can create any other parallel component). The visibility of component references shall follow the same scope rules as that of variables and in order to reference components outside their scope of creation the component reference shall be passed as a parameter or as a field in a message.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clauses 5 and 21 and shown in table 16, the following restrictions apply:

a) The name given by the first *Expression* shall be a **charstring** value and when assigned it shall appear as the first argument of the **create** function.

b) The host id given by the second *Expression* shall be a **charstring** value and, when assigned, it shall appear as the second argument of the **create** function.

***Examples***

 // This example declares variables of type MyComponentType, which is used to store the

 // references of newly created component instances of type MyComponentType which is the

 // result of the create operations. An associated name is allocated to some of the created

 // component instances.

 :

 **var** MyComponentType v\_myNewComponent;

 **var** MyComponentType v\_myNewestComponent;

 **var** MyComponentType v\_myAliveComponent;

 **var** MyComponentType v\_myAnotherAliveComponent;

 **var** MyComponentType v\_myDeployedComponent;

 :

 v\_myNewComponent := MyComponentType.**create**;

 v\_myNewestComponent := MyComponentType.**create**("Newest");

 v\_myAliveComponent := MyComponentType.**create alive**;

 v\_myAnotherAliveComponent := MyComponentType.**create**("Another Alive") **alive**;

 v\_myDeployedComponent := MyComponentType.**create**(-, "Host4");

### 21.3.2 The Start component operation

The start operation is used to associate a behaviour to a component, which is then being executed by that component.

***Syntactical Structure***

*ObjectReference* "." **start** "(" ( *FunctionInstance | AltstepInstance* ) ")"

***Semantic Description***

Once a component has been created and connected, behaviour has to be bound to this component and the execution of its behaviour has to be started. This is done by using the **start** operation (as component 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 component.

The **start** operation shall bind the required behaviour to the component. This behaviour is defined by reference to an already defined function or altstep.

An alive-type component may perform several behaviours in sequential order. Starting a second behaviour on a non-alive component or starting a behaviour on a component that is still running results in a test case error. If a behaviour is started on an alive-type component 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 component is stopped. This means no default is activated when a new behaviour is started on an alive-type component.

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 component can be accessed by a behaviour with a corresponding **runs on**-clause that is started on an alive-type component.

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** component operation, i.e. templates denoted by **return** keyword or **inout** and **out** parameters, have no effect when the started component terminates.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clauses 5 and 21 and shown in table 16, the following restrictions apply:

1. The *ObjectReference* shall be of component type and shall not resolve to a template.
2. The function or altstep invoked in a **start** component operation shall have a **runs on** definition referencing a component type that is compatible with the newly created component (see clause 6.3.2.7).
3. All formal parameters of the function or altstep invoked in a **start** component operation shall be of a component data type.
4. When used from inside a control behaviour the started behavior must also be a control behaviour.

NOTE 2: As **in** and **inout** ports starts listening when the component is created, at the moment, when it starts execution there may be messages in the incoming queues of such ports already waiting to be processed.

***Examples***

 **function** f\_myFirstBehaviour() **runs on** MyComponentType { … }

 **function** f\_mySecondBehaviour() **runs on** MyComponentType { … }

 **function** f\_myThirdBehaviour(out integer p\_p1, inout integer p\_p2) **runs** **on** MyComponentType { … }

 **altstep** a\_myFourthBehaviour() **runs on** MyComponentType { ... }

 :

 **var** MyComponentType v\_myNewPTC;

 **var** MyComponentType v\_myAlivePTC;

 **var integer** v\_int := 0;

 :

 v\_myNewPTC := MyComponentType.**create**; // Creation of a new non-alive test component.

 v\_myAlivePTC := MyComponentType.**create alive**; // Creation of a new alive-type test component

 :

 v\_myNewPTC.**start**(f\_myFirstBehaviour()); // Start of the non-alive component.

 v\_myNewPTC.**done**; // Wait for termination

 v\_myNewPTC.**start**(f\_mySecondBehaviour()); // Test case error

 :

 v\_myAlivePTC.**start**(f\_myFirstBehaviour()); // Start of the alive-type component

 v\_myAlivePTC.**done**; // Wait for termination

 v\_myAlivePTC.**start**(f\_mySecondBehaviour()); // Start of the next function on the same component

 :

 v\_myAlivePTC.**start**(f\_myThirdBehaviour(-,v\_int)); // v\_int will not be changed by the function

 v\_myAlivePTC.**done**;

 v\_myAlivePTC.**start**(a\_myFourthBehaviour()); // Direct start of an altstep behaviour<>

### 21.3.3 The Stop component operation

The stop component operation is used to stop the execution of a component by itself or by another component.

***Syntactical Structure***

**stop** |

( ( *ObjectReference* | **mtc** | **self** ) "." **stop** ) |

( **all** **component** "." **stop** )

***Semantic Description***

By using the **stop** component statement a component can stop the execution of its own currently running behaviour or the execution of the behaviour running on another test component. If a component does not stop its own behaviour, but the behaviour running on another component in the test system, the component to be stopped has to be identified by using its component reference. A component can stop its own behaviour by using a simple **stop** execution statement (see clause 19.9) or by addressing itself in the **stop** operation, e.g. by using the **self** operation.

NOTE 1: While the **create**, **start**, **running,** **done** and **killed** operations can be used for PTC(s) and PCCs only, the **stop** operation can also be applied to the MTC.

Stopping a component is the explicit form of terminating the execution of the currently running behaviour. A component behaviour terminates also by completing its execution upon reaching the end of the behaviour that is started on this component or by an explicit **return** statement. This termination is also called implicit stop. The implicit stop has the same effects as an explicit stop, i.e.for PTCs the global verdict is updated with the local verdict of the stopped test component (see clause 24).

If the stopped test component is the MTC, resources of all existing PTCs shall be released, the PTCs shall be removed from the test system and the test case shall terminate (see clause 26.1). When a control component is stopped while it is executing a testcase, the MTC of that testcase is stopped.

Stopping a non-alive-type component (implicitly or explicitly) shall destroy it and all resources associated with the component shall be released.

Stopping an alive-type component shall stop the currently running behaviour only but the component continues to exist and can execute new behaviour (started on it using the **start** operation). Stopping an alive-type component means that all variables, timers and ports declared in the component type definition of the alive-type component keep their value, contents or state. Furthermore, the local verdict of the component keeps its value. In contrast to that, all active defaults are automatically deactivated when the alive-type component is stopped. The component shall be left in a consistent state after stopping its behaviour.

For example, if the behaviour of an alive-type component is stopped during assigning a new value to an already bound variable, the variable shall remain bound after the component is stopped (with the old or the new value). Similarly, if the component is stopped during re-starting an already running timer, the timer shall be left in the running state after termination of the behaviour.

The **all** keyword can be used by the MTC only in order to stop all running PTCs but the MTC itself. The all component construct can also be used by the master control component to stop all running parallel control components but itself.

NOTE 2: A PTC can stop the test case execution by stopping the MTC.

NOTE 3: The concrete mechanism for stopping PTCs is outside the scope of the present document.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clauses 5 and 21 and shown in table 16, the following restrictions apply:

1. The *ObjectReference* shall be of component type and shall not resolve to a template.

***Examples***

EXAMPLE 1: Stopping another test component and a test component by itself

 **var** MyComponentType v\_myComp := MyComponentType.**create**; // A new test component is created

 v\_myComp.**start**(f\_compBehaviour()); // The new component is started

 :

 **if** (v\_date == "1.1.2005") {

 v\_myComp.**stop;** // The component "v\_myComp" is stopped

 }

 :

 **if** (v\_a < v\_b ) {

 :

 **self**.**stop;** // The test component that is currently executing stops its own behaviour

 }

 :

 **stop** // The test component stops its own behaviour

EXAMPLE 2: Stopping all PTCs by the MTC

 **all component**.**stop** // The MTC stops all PTCs of the test case but not itself.

### 21.3.4 The Kill component operation

The **kill** component operation is used to destroy a component by itself or by another component. Kill and stop on a non-alive component have the same results, while they differ for alive components: stopping an alive components stops the behaviour only, the component continues to exist. Killing a component destroys the component.

***Syntactical Structure***

**kill** |

( ( *ObjectReference* | **mtc** | **self** ) "." **kill** ) |

( **all** **component** "." **kill** )

***Semantic Description***

The **kill** operation applied on a component stops the execution of the currently running behaviour - if any - of that component and frees all resources associated to it (including all port connections of the killed component) and removes the component from the test system. The **kill** operation can be applied on the current component itself by a simple **kill** statement or by addressing itself using the **self** operation in conjunction with the kill operation. The **kill** operation can also be applied to another component. In this case the component to be killed shall be addressed using its component reference. If the **kill** operation is applied on the MTC, e.g. **mtc.kill**, it terminates the test case. If the kill operation is applied to a control component while it is executing a testcase, the mtc of that testcase will be stopped.

The **all** keyword can be used by the MTC only in order to stop and kill all running PTCs but the MTC itself. The all component construct can also be used by the main control component to stop and kill all running parallel control components but itself.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5 and 21 and shown in table 16, the following restrictions apply:

1. The *ObjectReference* shall be of component type and shall not resolve to a template.

***Examples***

EXAMPLE 1: Killing another test component and a test component by itself

 **var** PTCType v\_myAliveComp := PTCType.**create alive**; // Create an alive-type test component

 v\_myAliveComp.**start**(f\_myFirstBehaviour()); // The new component is started

 v\_myAliveComp.**done**; // Wait for termination

 v\_myAliveComp.**start**(f\_mySecondBehavior()); // Start the component a 2nd time

 v\_myAliveComp.**done**; // Wait for termination

 v\_myAliveComp.**kill**; // Free its resources

EXAMPLE 2: Killing all PTCs by the MTC

 **all component**.**kill**; // The MTC stops all (alive-type and normal) PTCs of the test case first
 // and frees their resources.

### 21.3.5 The Alive operation

The **alive** operation is a Boolean operation that checks whether a component has been created and is ready to execute or is executing already a behaviour.

***Syntactical Structure***

( *ObjectReference* |

**any** **component** |

**all** **component** |

**any from** ComponentArrayRef ) "." **alive**

[ **"->" @index value** ValueRef]

***Semantic Description***

Applied on a normal parallel component, the **alive** operation returns true if the component is inactive or running a behaviour and false otherwise. Applied on an alive-type parallel component, the operation returns true if the component is inactive, running or stopped. It returns false if the component has been killed. Applied on the **mtc** of the main control component the operation returns **true**.

The **alive** operation can be used similar to the **running** operation (see clause 21.3.6). In particular, in combination with the **all** keyword it returns true if all (alive-type or normal) PTCs are alive inside a testcase behaviour or if all parallel control components are alive in a control behaviour.

The **alive** operation used in combination with the **any** keyword returns true if at least one PTC is alive in a testcase behaviour or at least one PCC is alive in a control behaviour.

When the **any from** component array notation is used, the components from the referenced array are iterated over and individually checked for being inactive or running a function from innermost to outermost dimension from lowest to highest index for each dimension. The first component to be found being inactive or running a behaviour causes the alive operation to return the **true** value. The index of the first component found alive can optionally be assigned to an integer variable for single-dimensional component arrays or to an integer array or record of integer variable for multi‑dimensional component arrays.

When used from a control behaviour, the all and any component contructs are only in reference to created parallel control components and do not include the test components of test cases these control components are executing.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clauses 5 and 21 and shown in table 16, the following restrictions apply:

1. The *ObjectReference* shall be of component type and shall not resolve to a template.
2. The *ComponentArrayRef* shall be a reference to a completely initialized component array.
3. The index redirection shall only be used when the operation is used on an **any from** component array construct.
4. If the index redirection is used for single-dimensional component arrays, the type of the integer variable shall allow storing the highest index of the respective array.
5. If the index redirection is used for multi-dimensional component arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective array, and its type shall allow storing the highest index (from all dimensions) of the array.
6. If a variable referenced in the **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **alive** operation, i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **alive** operation.

***Examples***

 pTC1.**done**; // Waits for termination of the component

 **if** (pTC1.**alive**) { // If the component is still alive …

 pTC1.**start**(f\_anotherFunction()); // … execute another function on it.

### 21.3.6 The Running operation

The **running** operation is a Boolean operation that checks whether a component is already executing a behaviour.

***Syntactical Structure***

( *ObjectReference* |

**any** **component** |

**all** **component** |

**any from** ComponentArrayRef ) "." **running**

[ "->" **@index value** ValueRef]

***Semantic Description***

The **running** operation allows behaviour executing on a component to ascertain whether behaviour running on a different component has completed. The running operation returns **true** for the **mtc**, the main control component and parallel componets that have been started but not yet terminated or stopped. It returns **false** otherwise. The **running** operation is considered to be a **boolean** expression and, thus, returns a **boolean** value to indicate whether the specified component (or all components) has terminated. In contrast to the **done** operation, the **running** operation can be used freely in **boolean** expressions.

When the **all** keyword is used with the **running** operation inside a testcase behaviour, it will return **true** if all PTCs started but not stopped explicitly by another component are executing their behaviour. When it is used inside a control behaviour, it will return true if all PCCs started but not stopped explicitly by another component are executing their behaviour. Otherwise it returns **false**.

NOTE: The difference between the **running** operation applied to a single ptc and the usage of the **all** keyword leads to the situation that **ptc.running** is **false** if the ptc has never been started but **all component.running** is **true** at the same time as it considers only those components that ever have been started.

When the **any** keyword is used with the **running** operation inside a testcase behaviour, it will return **true** if at least one PTC is executing its behaviour. When used inside a control behaviour, it will return true if at least one PCC is executing its behaviour. Otherwise it returns **false**.

When the **any from** component array notation is used, the components from the referenced array are iterated over and individually checked for executing currently from innermost to outermost dimension from lowest to highest index for each dimension. The first component to be found executing causes the running operation to succeed. The index of the matched component can optionally be assigned to an integer variable for single-dimensional arrays or to an integer array or record of integer variable for multi-dimensional component arrays.

When used from a control behaviour, the all and any component contructs are only in reference to created parallel control components and do not include the test components of test cases these control components are executing.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clauses 5 and 21 and shown in table 16, the following restrictions apply:

1. The *ObjectReference* shall be of component type and shall not resolve to a template.
2. The *ComponentArrayRef* shall be a reference to a completely initialized component array.
3. The index redirection shall only be used when the operation is used on an **any from** component array construct.
4. If the index redirection is used for single-dimensional component arrays, the type of the integer variable shall allow storing the highest index of the respective array.
5. If the index redirection is used for multi-dimensional component arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective array, and its type shall allow storing the highest index (from all dimensions) of the array.
6. If a variable referenced in the **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **running** operation. Later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **running** operation.

***Examples***

 **if** (pTC1.**running**) // usage of running in an if statement

 {

 // do something!

 }

 **while** (**all** **component**.**running** != **true**) { // usage of running in a loop condition

 f\_mySpecialFunction()

 }

### 21.3.7 The Done operation

The **done** operation allows behaviour executing on a component to ascertain whether the behaviour running on a different component has completed. In addition, the **done** operation allows to retrieve the final local verdict of completed test components, i.e., the value of the local verdict at the time of test component completion.

***Syntactical Structure***

[ @nodefault ] ( *ObjectReference* |

**any** **component** |

**all** **component** |

**any from** ComponentArrayRef ) "." **done**[ "->"[ **value** ValueRef] [ **@index value** ValueRef] ]

***Semantic Description***

The **done** operation shall be used in the same manner as a receiving operation or a **timeout** operation. This means it shall not be used in a **boolean** expression, but it can be used to determine an alternative in an **alt** statement or as stand-alone statement in a behaviour description. In the latter case a **done** operation is considered to be a shorthand for an **alt** statement with the **done** operation as the only alternative. If the **@nodefault** modifier is placed before a stand-alone **done** operation, the implicit **alt** statement also contains the **@nodefault** modifier.

When the **done** operation is applied to a component, it matches only if the behaviour of that component has been stopped (implicitly or explicitly) or the component has been killed. Otherwise, the match is unsuccessful.

NOTE 1: The execution of a **done** operation does not change the state of the test component. Consecutive **done** operations applied to the same test component will give the same result as long as the test component does not change its state (see clause F.1.2).

When the **done** operation is applied to a PTC and matches, the final local verdict of the PTC can be retrieved and stored in a variable of the type **verdicttype**. This is denoted by the symbol '**->**' the keyword **value** followed by the name of the variable into which the verdict is stored.

When the **all** keyword is used with the **done** operation inside a testcase behaviour, it matches if no one PTC is executing its behaviour. It also matches if no PTC has been created. When the **all** keyword is used with the **done** operation inside a control behaviour, it matches if no one PCC is executing its behaviour. It also matches if no PCC has been created.

NOTE 2: The difference between the **done** operation applied to a single ptc and the usage of the **all** keyword leads to the situation that **ptc.done** does not match if the ptc has never been started but **all component.done** matches at the same time as it considers only those components that ever have been started.

When the **any** keyword is used with the **done** operation inside a testcase behaviour, it matches if at least the behaviour of one PTC has been stopped or killed. If used inside a control behaviour, it matches if at least the behaviour of one PCC has been stopped or killed. Otherwise, the match is unsuccessful.

NOTE 3: Stopping the behaviour of a non-alive component also results in removing that component from the test system, while stopping an alive-type component leaves the component alive in the test system. In both cases the **done** operation matches.

When the **any from** component array notation is used, the components from the referenced array are iterated over and individually checked for being stopped or killed from innermost to outermost dimension from lowest to highest index for each dimension. The first component to be found stopped or killed causes done operation to succeed. The index of the matched component can optionally be assigned to an integer variable for single-dimensional arrays or to an integer array or record of integer variable for multi-dimensional component arrays.

When used from a control behaviour, the all and any component contructs are only in reference to created parallel control components and do not include the test components of test cases these control components are executing.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clauses 5 and 21 and shown in table 16, the following restrictions apply:

1. The **done** operation can be used for PTCs only.
2. The *ObjectReference* followed by the **done** keyword, i.e. used for identifying a specific PTC, shall be of a component type and shall not resolve to a template.
3. The *ComponentArrayRef* shall be a reference to a completely initialized component array.
4. The variable used in the (optional) **value** clause for storing the final local verdict of a PTC shall be of the type **verdicttype**.
5. The (optional) **value** clause for storing the final local verdict of a PTC shall not be used in combination with **all component** or **any component**.
6. The index redirection shall only be used when the operation is used on an **any from** component array construct.
7. If the index redirection is used for single-dimensional component arrays, the type of the integer variable shall allow storing the highest index of the respective array.
8. If the index redirection is used for multi-dimensional component arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective array, and its type shall allow storing the highest index (from all dimensions) of the array.
9. If a variable referenced in the **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **done** operation. Later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **done** operation.
10. The **@nodefault** modifier is allowed only in stand-alone **done** statements.
11. The verdict value redirect shall not be used inside control behaviour.

***Examples***

 // Use of done in alternatives

 **alt** {

 [] myPTC.**done** {

 **setverdict**(**pass**)

 }

 [] **any port**.**receive** {

  **repeat**

 }

 }

 **var** MyComp v\_c := MyComp.**create** **alive**;

 v\_c.**start**(f\_myPTCBehaviour());

 :

 v\_c.**done**;

 // matches as soon as the function f\_myPTCBehaviour (or function/altstep called by it) stops

 v\_c.**done**;

 // matches again, even if the component has not been started again

 if(v\_c.**running**) {v\_c.**done**}

 // in case that some other component has started v\_c in the meantime

 // done here matches the end of the next behaviour only, not the previous one

 // the following done as stand-alone statement:

 @nodefault **all component**.**done**;

 // has the following meaning:

 **alt** @nodefault {

 [] **all component**.**done** {}

 }

 // and thus, blocks the execution until all parallel test components have terminated while

 // ignoring all activated default alternatives

 // Retrieving and using the final local verdict of a completed PTC

 **var** MyComp v\_myPTC := MyPTC.**create** **alive**;

 **var verdicttype** v\_myPTCverdict := **none**;

 v\_myPTC.**start**(f\_myPTCBehaviour());

 :

 **alt** {

 [] v\_myPTC.**done** -> **value** v\_myPTCverdict{

 **if** (v\_myPTCverdict == **fail**) {

 **setverdict**(**fail**);

 **stop**;

 }

 **else** {

 **setverdict** (**pass**);

 }

 }

 [] **any port**.**receive** {

  **repeat**

 }

 }

### 21.3.8 The Killed operation

The **killed** operation allows to ascertain whether a different component is alive or has been removed from the test system. In addition, the **killed** operation allows to retrieve the final local verdict of killed test components, i.e., the value of the local verdict at the time when the test component was killed.

***Syntactical Structure***

[ @nodefault ] ( *ObjectReference* |

**any** **component** |

**all** **component** |

**any from** ComponentArrayRef ) "." **killed**

[ "->"[ **value** ValueRef] [ **@index value** ValueRef] ]

***Semantic Description***

The **killed** operation shall be used in the same manner as receiving operations. This means it shall not be used in **boolean** expressions, but it can be used to determine an alternative in an **alt** statement or as a stand-alone statement in a behaviour description. In the latter case a **killed** operation is considered to be a shorthand for an **alt** statement with the **killed** operation as the only alternative. If the **@nodefault** modifier is placed before a stand-alone **killed** operation, the implicit **alt** statement also contains the **@nodefault** modifier.

NOTE 1: When checking normal components a killed operation matches if it stopped (implicitly or explicitly) the execution of its behaviour or has been **killed** explicitly, i.e. the operation is equivalent to the **done** operation (see clause 21.3.7). When checking alive-type components, however, the **killed** operation matches only if the component has been killed using the **kill** operation. Otherwise the **killed** operation is unsuccessful.

NOTE 2: The execution of a **killed** operation does not change the state of the component. Consecutive **killed** operations applied to the same component will give the same result as long as the component does not change its state (see clause F.1.2).

When the **all** keyword is used with the **killed** operation inside testcase behaviour, it matches if all PTCs of the test case have ceased to exist. It also matches if no PTC has been created. When the **all** keyword is used with the **killed** operation inside control behaviour, it matches if all PCCs have ceased to exist. It also matches if no PCC has been created.

When the **killed** operation is applied to a PTC and matches, the final local verdict of that PTC can be retrieved and stored in a variable of the type **verdicttype**. This is denoted by the symbol '**->**' the keyword **value** followed by the name of the variable into which the verdict is stored.

When the **any** keyword is used with the **killed** operation, it matches if at least one PTC ceased to exist. Otherwise, the match is unsuccessful.

When the **any from** component array notation is used, the components from the referenced array are iterated over and individually checked for being killed from innermost to outermost dimension from lowest to highest index for each dimension. The first component to be found killed causes the killed operation to succeed. The index of the matched component can optionally be assigned to an integer variable for single-dimensional component arrays or to an integer array or record of integer variable for multi-dimensional component arrays.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clauses 5 and 21 and shown in table 16, the following restrictions apply:

1. The **killed** operation can be used for PTCs only.
2. The *ObjectReference* followed by the **killed** keyword, i.e. used for identifying a specific PTC, shall be of a component type and shall not resolve to a template.
3. The *ComponentArrayRef* shall be a reference to a completely initialized component array.
4. The variable used in the (optional) **value** clause for storing the final local verdict of a PTC shall be of the type **verdicttype**.
5. The (optional) **value** clause for storing the final local verdict of a PTC shall not be used in combination with **all component** or **any component**.
6. The index redirection shall only be used when the operation is used on an **any from** component array construct.
7. If the index redirection is used for single-dimensional component arrays, the type of the integer variable shall allow storing the highest index of the respective array.
8. If the index redirection is used for multi-dimensional component arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective array, and its type shall allow storing the highest index (from all dimensions) of the array.
9. If a variable referenced in the **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **killed** operation i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **killed** operation.
10. The **@nodefault** modifier is allowed only in stand-alone **killed** statements.
11. The verdict value redirect shall not be used inside control behaviour.

***Examples***

 **var** MyPTCType v\_ptc := MyPTCType.**create** **alive**; // create an alive-type test component

 **timer** t\_T:= 10.0; // create a timer

 t\_T.**start**; // start the timer

 v\_ptc.**start**(f\_myTestBehavior()); // start executing a function on the PTC

 **alt** {

 [] v\_ptc.**killed** { // if the PTC was killed during execution …

 t\_T.**stop**; // … stop the timer and …

 **setverdict**(**inconc**); // … set the verdict to 'inconclusive'

 }

 [] v\_ptc.**done** { // if the PTC terminated regularly …

 t\_T.**stop**; // … stop the timer and …

 v\_ptc.**start**(f\_anotherFunction()); // … start another function on the PTC

 }

 [] t\_T.**timeout** { // if the timeout occurs before the PTC stopped

 v\_ptc.**kill**; // … kill the PTC and …

 **setverdict**(**fail**); // … set the verdict to 'fail'

 }

 }

 // Retrieving and using the final local verdict of a killed PTC

 **var** MyComp v\_myPTC := MyPTC.**create** **alive**;

 **var** **verdicttype** v\_myPTCverdict := **none**;

 v\_myPTC.**start**(f\_myPTCBehaviour());

 :

 **alt** {

 [] v\_myPTC.**done** { // expected termination

 **setverdict** (**pass**);

 }

 }

 [] v\_myPTC.**killed** -> **value** v\_myPTCverdict{

 **if** (v\_MyPTCverdict == **none**) { // v\_myPTC killed before verdict assignment

 **setverdict**(**fail**);

 **stop**;

 }

 **else** {

 **setverdict** (**inconc**); // further analysis is needed

 **stop**;

 }

 }

 [] **any port**.**receive** {

  **repeat**

 }

 }

### 21.3.9 Summary of the use of any and all with components

The keywords **any** and **all** may be used with configuration operations as indicated in table 21.

Table 21: Any and All with components

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Allowed | Example | Comment |
|  | **any (see note)** | all (see note) |  |  |
| **create** |  |  |  |  |
| **start** |  |  |  |  |
| **running** | Yes but from MTC or MCC only | Yes but from MTC or MCC only | **any component.running;****all component.running;** | Is there any parallel component performing behaviour?Are all parallel components performing behaviour? |
| **alive** | Yes but from MTC or MCC only | Yes but from MTC or MCC only | **any component.alive;****all component.alive;** | Is there any alive parallel component?Are all parallel components alive? |
| **done** | Yes but from MTC or MCC only | Yes but from MTC or MCC only | **any component.done;****all component.done;** | Is there any parallel component that completed execution?Did all parallel components complete their execution? |
| **killed** | Yes but from MTC or MCC only | Yes but from MTC or MCC only | **any component.killed;****all component.killed;** | Is there any parallel component that ceased to exist?Did all parallel components cease to exist? |
| **stop** |  | Yes but from MTC or MCC only | **all component.stop;** | Stop the behaviour on all parallel components. |
| **kill** |  | Yes but from MTC or MCC only | **all component.kill;** | Kill all parallel components, i.e. they cease to exist. |
| NOTE: **any** and **all** refer to parallel components only, i.e. the MTC/MCC is not considered. |

### 21.3.10 The Call test component behaviour operation

The call operation is used start a test behaviour on a test component and wait until that behaviour has terminated.

***Syntactical Structure***

*ObjectReference* "." **call** "(" ( *FunctionInstance | AltstepInstance* )

*[*"," *SimpleExpression* ] ")"

[ "->" [**value** *Ref*] [**verdict** Ref] ]

[ **catch** "(" **timeout** ")" *StatementBlock* ]

[ catch "(" **stop** ")" *StatementBlock* ]

***Semantic Description***

Similar to the start operation on test components which is not blocking, the blocking **call** operation implicitly uses a **start** operation, but waits until either the started behaviour has terminated or some timeout has occurred.

A timeout duration in seconds can be given explicitly in the form of a *SimpleExpression* as an additional parameter to the call operation. If no timeout duration is given, an infinite timeout duration is used.

The actions taken by the call operation are dependent on whether the execution of the started behaviour is complete or incomplete. Complete execution occurs when the started function is terminated by executing a return statement or if it reaches the end of the function body. If the started behaviour is terminated for any other reason, the execution is incomplete.

If the incomplete execution occurs because the called component was stopped or killed and a **catch stop** clause is added to the call operation, the *StatementBlock* of that clause is executed before the call operation terminates.

If the started behaviour does not terminate in the given timeout duration and a **catch timout** clause is added to the call operation, the called component is implicitly stopped and the *StatementBlock* of the catch timeout clause is executed before the call operation terminates.

In all other cases when the execution is incomplete, the call operation ends with a test case error.

After complete execution of the started behaviour, the **out** and **inout** actual parameters given to the actual parameter list of the called function or altstep instance will be updated in the same manner as if it was a normal function/altstep invocation.

Additionally, a redirect clause can be added to the operation which allows assignment of the return result (in case that the called function has a return clause) to a variable via the **value** clause and also the assignment of the termination verdict of the called component via the **verdict** clause.

In all cases of incomplete execution, the variables referenced in the value and verdict clause or in **out** and **inout** actual parameters will stay unchanged and no assignment will be made.

If the called component is not created alive and has already been started or called once or if it has been killed, additional call operations are not allowed.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clauses 5 and 21 and shown in table 16, the following restrictions apply:

1. The *ObjectReference* shall be of a component type.
2. The function or altstep invoked in a **call** test component operation shall have a **runs on** definition referencing a component type to which the called component is compatible (see clause 6.3.2.7).
3. All formal parameters of the function or altstep invoked in a **call** test component operation shall be of a component data type.
4. The return value of the function invoked from a **call** test component operation shall be of a component data type.
5. The optional *SimpleExpression* representing the timer value shall be of a float type.
6. The optional **catch** **timeout** clause may be present only if the timeout value has been defined.
7. The variable in the value clause shall be compatible with the return value of the invoked function.
8. The variable in the verdict clause shall be of type verdicttype.

***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

 return integer { … }

 **altstep** a\_myFourthBehaviour() **runs on** MyComponentType { ... }

 :

 **var** MyComponentType v\_myNewPTC;

 **var** MyComponentType v\_myAlivePTC;

 **var integer** v\_out, v\_inout := 0, v\_result;

 :

 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.**call**(f\_myFirstBehaviour()); // Call to the non-alive component.

 v\_myNewPTC.**call**(f\_mySecondBehaviour()); // Test case error

 :

 v\_myAlivePTC.**call**(f\_myFirstBehaviour()); // Call to the alive-type component

 v\_myAlivePTC.**call**(f\_mySecondBehaviour()); // Another call to the same component

 :

 v\_myAlivePTC.**call**(f\_myThirdBehaviour(v\_out,v\_inout)) // v\_out/v\_inout can be changed

-> value v\_result verdict v\_verdict; // v\_result/v\_verdict are assigned on successful
 // termination

v\_myAlivePTC.**call**(a\_myFourthBehaviour()); // Direct call of an altstep behaviour

# 26 Module control

## 26.0 General

Test cases are defined in the module definitions part while the control functions manage their execution. The statements and operations that can be used in control behaviour are summarized in table 31.

Table 31: Overview of TTCN‑3 statements and operations in module control

| Statement | Associated keyword or symbol |
| --- | --- |
| Assignments  | **:=** |
| If-else  | **if (…) {…} else {…}** |
| Select case | **select case (…) { case (…) {…} case else {…}}** |
| For loop  | **for (…) {…}** |
| While loop  | **while (…) {…}** |
| Do while loop | **do {…} while (…)** |
| Label and Goto | **label / goto** |
| Stop execution | **stop** |
| Leaving a loop, alt or interleave | **break** |
| Next iteration of a loop | **continue** |
| Logging  | **log** |
| Alternative behaviour | **alt {…}**  |
| Re-evaluation of alternative behaviour | **repeat** |
| Interleaved behaviour | **interleave {…}** |
| Activate a default | **activate** |
| Deactivate a default | **deactivate** |
| Start timer | **start** |
| Stop timer | **stop** |
| Read elapsed time | **read** |
| Check if timer or component running | **running** |
| Timeout event | **timeout** |
| Stimulate an (SUT) action externally | **action** |
| Execute test case | **execute** |
| Create component | **create** |
| Component control operations | **start, stop, kill** |
| Component events | **done, killed** |
| Configuration operations | **connect, disconnect, map, unmap** |
|  |  |
| Port sending operations | **send, call, reply, raise** |
| Port control operations | **halt, clear, start** |
| Port receiving operations | **receive, getcall, getreply, catch** |
|  |

## 26.1 The Execute statement

Test cases are executed with an **execute** statement in the module control.

***Syntactical Structure***

**execute** "(" *TestcaseRef* "(" [ { *ActualPar* [","] } ] ")" [ "," *TimerValue* [ "," *HostId* ] ] ")"

***Semantic Description***

The **execute** statement is used to start test cases (see clause 27.1) in control behaviour. The result of an executed test case is always a value of type **verdicttype**. Every test case shall contain one and only one MTC the type of which is referenced in the header of the test case definition. The behaviour defined in the test case body is the behaviour of the MTC.

When a test case is invoked the MTC is created, the ports of the MTC and the test system interface are instantiated and the behaviour specified in the test case definition is started on the MTC. All these actions shall be performed implicitly i.e. without explicit **create** and **start** operations.

**Test case start**

A test case is called using an **execute** statement. As the result of the execution of a test case, a test case verdict of either **none**, **pass**, **inconc**, **fail** or **error** shall be returned and may be assigned to a variable for further processing.

Optionally, the **execute** statement allows supervision of a test case by means of a timer duration.

Also optionally, the execute statement allows deployment of the MTC to a specific host before starting the execution. The host is identified by means of a host id.

**Test case parameterization and configuration**

All variables (if any) visible in the scope unit where the **execute** statement is used shall be passed into the test case by parameterization if they are to be used in the behaviour definition of that test case, i.e. TTCN‑3 does not support global variables of any kind.

At the start of each test case, the test configuration shall be reset. This means that all components and ports conducted by **create**, **connect,** etc. operations in a previous test case were destroyed when that test case was stopped (hence are not "visible" to the new test case).

**Test case termination**

A test case terminates with the termination of the MTC. On termination of the MTC (explicitly or implicitly), all running parallel test components shall be removed by the test system.

NOTE 1: The concrete mechanism for stopping all PTCs is tool specific and therefore outside the scope of the present document.

The final verdict of a test case is calculated based on the final local verdicts of the different test components according to the rules defined in clause 24.1. The actual local verdict of a test component becomes its final local verdict when the test component terminates itself or is stopped by itself, another test component or by the test system.

NOTE 2: To avoid race conditions for the calculation of test verdicts due to the delayed stopping of PTCs, the MTC should ensure that all PTCs have stopped (by means of the **done** or **killed** statement) before it stops itself.

**Test case timer**

Timer may be used to supervise the execution of a test case. This may be done using an explicit timeout in the **execute** statement. If the test case does not end within this duration, the result of the test case execution shall be an error verdict and the test system shall terminate the test case. The timer used for test case supervision is a system timer and need not be declared or started.

**Host id**

A host id can be used to give a specific deployment location to the test system where the MTC shall be started and execute its behaviour. If a host id is provided, the execute statement shall end with a test case error if the MTC cannot be deployed on the specified host.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 16, the following restrictions apply:

a) The *TimerValue* shall resolve to a non-negative numerical float value (i.e. the value shall be greater or equal 0.0, infinity and not\_a\_number are disallowed).

b) When the corresponding formal parameter is not of template type *TemplateInstance* shall resolve to an *Expression*.

c) The execute statement shall not be called from within an existing executing test behaviour chain called from a test case, i.e. test cases can only be executed from a control behaviour.

d) The *HostId* parameter shall resolve to a **charstring** value.

***Examples***

EXAMPLE 1: Test case execution without keeping the test case verdict:

 **execute**(TC\_MyTestCase1()); // executes TC\_MyTestCase1, without storing the

 // returned test verdict and without time supervision

EXAMPLE 2: Test case execution with keeping the test case verdict:

 v\_myVerdict := **execute**(TC\_MyTestCase2()); // executes TC\_MyTestCase2 and stores the resulting

 // verdict in variable v\_myVerdict

EXAMPLE 3: Test case timer:

 v\_myVerdict := **execute**(TC\_MyTestCase3(),5E-3);

 // executes TC\_MyTestCase3 and stores the resulting verdict in variable v\_myVerdict.

 // If the test case does not terminate within 5ms, v\_myVerdict will get the value 'error'

EXAMPLE 4: Host id:

 v\_myVerdict := **execute**(TC\_MyTestCase3(), -, "Host1");

 // executes TC\_MyTestCase3 with unlimited time with MTC deployed to 'Host1'

## 26.2 Test suite execution

TTCN-3 test suite execution is controlled by the the module control function. The module control function defines, in which order, sequence, loop, under which preconditions, and with which parameters test cases are to be executed.

**Execution and control component**

The module control function is an entry point for execution of a TTCN-3 test suite. If the function contains formal parameters, their actual values shall be provided. When the control function is started, the TE creates a test component called control component. The component contains variables, constants, templates and times important for controlling of the execution of the test suite. The created control component is of the type specified in the **runs on** clause of the module control function. If the **runs** **on** clause is missing, an empty control component is created.

**Sequence of test cases**

Program statements specify such things like the order in which test cases are to be executed or the number of times a test case should run.

If no programming statements are used then, by default, the test cases are executed in the sequential order in which they appear in the module control function.

NOTE: This does not preclude the possibility that certain tools may wish to override this default ordering to allow a user or tool to select a different execution order.

Timer operations may also be used explicitly to control test case execution.

**Selection/deselection of test cases**

The selection and deselection of test cases can also be used to control the execution of test cases.

There are different ways in TTCN‑3 to select and deselect test cases. For example, boolean expressions may be used to select and deselect which test cases are to be executed. This includes, of course, the use of functions that return a **boolean** value.

Another way to execute test cases as a group is to collect them in a control function and invoke that function from the module control function.

As a test case returns a single value of type **verdicttype**, it is also possible to control the order of test case execution depending on the outcome of a test case. The use of the TTCN‑3 verdicttype is another way to select test cases.

***Examples***

EXAMPLE 1: Test case execution in a loop:

 **module** MyTestSuite () {

 :

 **control** {

 :

 // Do this test 10 times

 v\_count:=0;

 **while** (v\_count < 10)

 { **execute** (TC\_MySimpleTestCase1());

 V\_count := v\_count+1;

 }

 }

 }

EXAMPLE 2: Test case execution controlled by a timer and a counter:

 // Example of the use of the running timer operation

 **while** (t­\_t1.**running** **or** v\_x<10) // Where t\_t1 is a previously started timer

 { **execute**(TC\_MyTestCase());

 v\_x := v\_x+1;

 }

 // Example of the use of the start and timeout operations

 **timer** t­\_t1:= 1.0;

 :

 **execute**(TC\_MyTestCase1());

 t­\_t1.**start**;

 t­\_t1.**timeout**; // Pause before executing the next test case

 **execute**(TC\_MyTestCase2());

EXAMPLE 3: Selection/deselection of test cases with Boolean expressions:

 **module** MyTestSuite () {

 :

 **control** {

 :

 **if** (f\_mySelectionExpression1()) {

 **execute**(TC\_MySimpleTestCase1());

 **execute**(TC\_MySimpleTestCase2());

 **execute**(TC\_MySimpleTestCase3());

 }

 **if** (f\_mySelectionExpression2()) {

 **execute**(TC\_MySimpleTestCase4());

 **execute**(TC\_MySimpleTestCase5());

 **execute**(TC\_MySimpleTestCase6());

 }

 :

 }

 }

EXAMPLE 4: Selection/deselection of test cases with functions:

 **function** f\_myTestCaseGroup1()

 { **execute**(TC\_MySimpleTestCase1());

 **execute**(TC\_MySimpleTestCase2());

 **execute**(TC\_MySimpleTestCase3());

 }

 **function** f\_myTestCaseGroup2()

 { **execute**(TC\_MySimpleTestCase4());

 **execute**(TC\_MySimpleTestCase5());

 **execute**(TC\_MySimpleTestCase6());

 }

 :

 **control**

 { **if** (f\_mySelectionExpression1()) { f\_myTestCaseGroup1(); }

 **if** (f\_mySelectionExpression2()) { f\_myTestCaseGroup2(); }

 :

 }

EXAMPLE 5: Selection/deselection of test cases based on test case verdicts:

 **if** ( **execute** (TC\_MySimpleTestCase()) == **pass** )

 { **execute** (TC\_MyGoOnTestCase()) }

 **else**

 { **execute** (TC\_MyErrorRecoveryTestCase()) };

## 26.3 Parallel control components

The same way that a testcase can create parallel test components, a control function can create parallel control components (PCCs). Every control component can only be started with a control function at a time.

Control components work the same way as test components, they can communicate over ports with each other and with the system under test. Once the main control component terminates, all PCCs are terminated as well.

The **all component** construct used in a control behavior references all PCCs. Only the main control component can wait for all PCCs to be done or stop/kill all PCCs.