ETSI ES 201 873-1 V4.11.1 (2019-04)

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-201873-1v4.11.1\_Core

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 2019.

All rights reserved.

**DECT**TM, **PLUGTESTS**TM, **UMTS**TM and the ETSI logo are trademarks of ETSI registered for the benefit of its Members.
**3GPP**TM and **LTE**TM 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.

### 16.2.1 Invoking altsteps

The invocation of an altstep is always related to an **alt** statement. The invocation may be done either implicitly by the default mechanism (see clause C.5) or explicitly by a direct call within an **alt** statement (see clause 20.2).

***Syntactical Structure***

*AltstepRef* "(" [ { *ActualPar* [","] } ] ")"

***Semantic Description***

The invocation of an altstep causes no new snapshot and the evaluation of the top alternatives of an altstep is done by using the actual snapshot of the **alt** statement from which the altstep was called.

NOTE 1: A new snapshot within an altstep will of course be taken, if within a selected top alternative a new **alt** statement is specified and entered.

For an implicit invocation of an altstep by means of the default mechanism, the altstep shall be activated as a default by means of an **activate** statement before the place of the invocation is reached.

An explicit call of an altstep within an **alt** statement looks syntactically like a function invocation as an alternative. When an altstep is called explicitly within an **alt** statement, the next alternative to be checked is the first alternative of the **altstep**. The alternatives of the **altstep** are checked and executed the same way as alternatives of an **alt** statement (see clause 20.1) with the exception that no new snapshot is taken when entering the **altstep**. An unsuccessful termination of the altstep (i.e. all top alternatives of the **altstep** have been checked and no matching branch is found) causes the evaluation of the next alternative or invocation of the default mechanism (if the explicit call is the last alternative of the **alt** statement). A successful termination may cause either the termination of the test component, i.e. the altstep ends with a **stop** statement, or a new snapshot and re-evaluation of the **alt** statement, i.e. the altstep ends with **repeat** (see clause 20.2) or a continuation immediately after the **alt** statement, i.e. the execution of the selected top alternative of the altstep ends with a **break** statement (see clause 19.12) or without explicit **repeat** or **stop**.

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

An **altstep** can also be called as a stand-alone statement in a TTCN‑3 behaviour description. In this case, the call of the **altstep** can be interpreted as shorthand for an **alt** statement with only one alternative describing the explicit call of the **altstep**. If the **@nodefault** modifier is placed before a stand-alone **altstep** call, the implicit **alt** statement also contains the **@nodefault** modifier.

***Restrictions***

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

a) When invoking an altstep, the compatibility of the test component type of the invoking test component and of the altstep runs on clause (as described in clause 6.3.3) need to be fulfilled.

b) Further restrictions on invoking altsteps in the activate statement are given in clause 20.5.2.

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

***Examples***

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

 :

 **var** **default** v\_myDefVarTwo := **activate**(a\_mySecondAltStep()); // Activation of an altstep as

 // default

 :

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

 :

 **alt** {

 [] pCO3.**receive** {

 …

 }

 [] a\_anotherAltStep(); // explicit call of altstep a\_anotherAltStep as an alternative

 // of an alt statement

 [] t\_myTimer.**timeout** {}

 }

EXAMPLE 3: Explicit, stand-alone invocation of an altstep

 // The statement

 a\_anotherAltStep(); // a\_anotherAltStep is assumed to be a correctly defined altstep

 //is a shorthand for

 **alt** {

 [] a\_anotherAltStep();

 }

## 20.2 The Alt statement

An alt statement expresses sets of possible alternatives that form a tree of possible execution paths.

***Syntactical Structure***

**alt** [ **@nodefault** ] "{"

 {

 "[" [ *BooleanExpression* ] "]"

 ( ( *TimeoutStatement* |

 *ReceiveStatement* |

 *TriggerStatement* |

 *GetCallStatement* |

 *CatchStatement* |

 *CheckStatement* |

 *GetReplyStatement* |

 *DoneStatement* |

 *KilledStatement* ) *StatementBlock* )

|

 ( *AltstepInstance* [ *StatementBlock* ] )

 }

 [ "[" **else** "]" *StatementBlock* ]

"}"

***Semantic Description***

The **alt** statement denotes branching of test behaviour due to the reception and handling of communication and/or timer events and/or the termination of parallel test components, i.e. it is related to the use of the TTCN‑3 operations **receive**, **trigger**, **getcall**, **getreply**, **catch**, **check**, **timeout, done** and **killed**. The **alt** statement denotes a set of possible events that are to be matched against a particular snapshot.

**Execution of alternative behaviour:**

When entering an **alt** statement, a snapshot is taken.

The alternative branches in the **alt** statement and the top alternatives of invoked altsteps and altsteps that are activated as defaults are processed in the order of their appearance. If several defaults are active, the reverse order of their activation determines the evaluation order of the top alternatives in the defaults. The alternative branches in active defaults are reached by the default mechanism described in clause 20.5. If the **alt** statement contains the **@nodefault** modifier, all active default alternatives are ignored for the execution of this **alt** statement.

The individual alternative branches are either branches that may be guarded by a Boolean expression or else-branches, i.e. alternative branches starting with [**else**].

Else-branches are always chosen and executed when they are reached (see below).

Branches that may be guarded by boolean expressions either invoke an altstep (*altstep-branch*), or start with a **done** operation (*done-branch*), a **killed** operation (*killed-branch*), **timeout** operation (*timeout-branch*) or a receiving operation (*receiving-branch*), i.e. **receive**, **trigger**, **getcall**, **getreply,** **catch** or a **check** operation. The evaluation of the Boolean guards shall be based on the snapshot. The Boolean guard is considered to be *fulfilled* if no Boolean guard is defined, or if the Boolean guard evaluates to **true**. The branches are processed and executed in the following manner.

An *altstep-branch* is selected if the Boolean guard is fulfilled. The selection of an *altstep-branch* causes the invocation of the referenced altstep, i.e. the altstep is invoked and the evaluation of the snapshot continues within the altstep. An altstep-branche may contain an optional statement block. The optional statement block shall be executed only, if an alternative of the altstep referenced in the altstep-branch has been selected and executed.

A *done-branch* is selected if the Boolean guard is fulfilled and if the specified test component is in the list of stopped components of the snapshot. The selection causes the execution of the statement block following the **done** operation. The **done** operation itself has no further effect.

A *killed-branch* is selected if the Boolean guard is fulfilled and if the specified test component is in the list of killed components of the snapshot. The selection causes the execution of the statement block following the **killed** operation. The **killed** operation itself has no further effect.

A *timeout-branch* is selected if the Boolean guard is fulfilled and if the specified timeout event is in the timeout-list of the snapshot. The selection causes execution of the specified **timeout** operation, i.e. removal of the timeout event from the timeout-list, and the execution of the statement block following the **timeout** operation.

A *receiving-branch* is selected if the Boolean guard is fulfilled and if the matching criteria of receiving operation is fulfilled by one of the messages, calls, replies or exceptions in the snapshot. The selection causes execution of the receiving operation, i.e. removal of the matching message, call, reply or exception from the port queue, maybe an assignment of the received information to a variable and the execution of the statement block following the receiving operation. In the case of the **trigger** operation the top message of the queue is also removed if the Boolean guard is fulfilled but the matching criteria is not. In this case the statement block of the given alternative is not executed.

NOTE 1: The TTCN‑3 semantics describe the evaluation of a snapshot as a series of indivisible actions of a test component. The semantics do not assume that the evaluation of a snapshot has no duration. During the evaluation of a snapshot, test components may stop, timers may timeout and new messages, calls, replies or exceptions may enter the port queues of the component However, these events do not change the actual snapshot and thus, are not considered for the snapshot evaluation.

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

If none of the alternative branches in the **alt** statement and top alternatives in the invoked altsteps and active defaults can be selected and executed, the **alt** statement shall be executed again, i.e. a new snapshot is taken and the evaluation of the alternative branches is repeated with the new snapshot. This repetitive procedure shall continue until either an alternative branch is selected and executed, or the test case is stopped by another component or by the test system (e.g. because the MTC is stopped) or with a dynamic error.

The test case shall stop and indicate a dynamic error if a test component is completely blocked. This means none of the alternatives can be chosen, no relevant test component is running, no relevant timer is running and all relevant ports contain at least one message, call, reply or exception that do not match.

NOTE 3: The repetitive procedure of taking a complete snapshot and re-evaluate all alternatives is only a conceptual means for describing the semantics of the **alt** statement. The concrete algorithm that implements this semantics is outside the scope of the present document.

**Selecting/deselecting an alternative:**

If necessary, it is possible to enable/disable an alternative by means of a Boolean expression placed between the ("[…]") brackets of the alternative.

**Else branch in alternatives:**

Any branch in an **alt** statement can be defined as an else branch by including the **else** keyword between the opening and closing brackets at the beginning of the alternative. The statement block of the else branch is always executed if no other alternative textually preceding the else branch has proceeded.

**Default mechanism:**

It should be noted that the default mechanism (see clause 20.5) is always invoked at the end of all alternatives unless the @nodefault modifier is present. If an **else** branch is defined, the default mechanism will never be called, i.e. active defaults will never be entered.

NOTE 4: It is also possible to use **else** in altsteps.

NOTE 5: It is allowed to use a **repeat** statement within an **else** branch.

NOTE 6: It is allowed to define more than one else branch in an alt statement or in an altstep, however always only the first else branch is executed.

**Re-evaluation of alt statements:**

The re-evaluation of an **alt** statement can be specified by using a **repeat** statement (see clause 20.3).

**Invocation of altsteps as alternatives:**

TTCN‑3 allows the invocation of altsteps as alternatives in **alt** statements (see clause 16.2.1). When an altstep is explicitly invoked as an alternative, the optional statement block following the altstep call shall also be executed.

**Continue execution after the alt statement:**

Behaviour execution continues with the statement following the **alt** statement when one of the branches of the **alt** or invoked defaults is selected and completely executed, or a branch of an **altstep** used in an altsteps-branch is selected and the branch and the optional statement block following the invoked altstep are completely executed.

Execution also continues with the statement following the **alt** statement if a **break** statement is reached in the statement block of the selected branch of an **alt** statement, of an **altstep** used in an altstep-branch, or of an **altstep** invoked as default.

The **alt** statement can also be left by using a **goto** statement in the selected branch of the **alt** (i.e. no branches of altsteps and defaults can be considered in this case), and execution continues with the statement following the label, **goto** is pointing to.

***Restrictions***

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

1. The open and close square brackets ("[…]") shall be present at the start of each alternative, even if they are empty. This not only aids readability but also is necessary to syntactically distinguish one alternative from another.
2. The evaluation of a Boolean expression guarding an alternative shall not have side effects. To avoid side effects that cause an inconsistency between the actual snapshot and the state of the component, the same restrictions as the restrictions for the initialization of local definitions within altsteps (clause 16.1.5) and the restrictions imposed on the contents of functions called from special places (clause 16.1.4) shall apply.
3. The evaluation of the event of an alt branch shall not have side effects. To avoid side effects that cause an inconsistency between the actual snapshot and the state of the component or introduce indeterminism in the evaluation of the following alt branches or the re-evaluation of the same alt branch, the restrictions imposed on the contents of functions called from special places (clause 16.1.4) shall apply to expressions occurring in the matching part of an alternative.
4. The evaluation of an altstep invoked from an alt branch, if none of the alternatives in the altstep is chosen, shall not have side effects. To avoid side effects the restrictions imposed on the contents of functions called from special places (clause 16.1.4) shall apply to the actual parameters of the invoked altstep.
5. Void.
6. An **alt** statement used inside control behaviour shall only contain **timeout** statements.

***Examples***

EXAMPLE 1: Nested alternatives

 **alt** {

 [] myPort.**receive** (mw\_myMessage) {

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

 t\_myTimer.**start**;

 **alt** {

 [] myPort.**receive** (mw\_mySecondMessage) {

 t\_myTimer.**stop**;

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

 }

 [] t\_myTimer.**timeout** {

 myPort.**send** (m\_myRepeat);

 t\_myTimer.**start**;

 a**l**t {

 [] myPort.**receive** (mw\_mySecondMessage) {

 t\_myTimer.**stop**;

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

 }

 [] t\_myTimer.**timeout** { **setverdict** (**inconc**) }

 [] myPort.**receive** { **setverdict** (**fail**) }

 }

 }

 [] myPort.**receive** { **setverdict** (**fail**) }

 }

 }

 [] t\_myTimer.**timeout** { **setverdict** (**inconc**) }

 [] myPort.**receive** { **setverdict** (**fail**) }

 }

EXAMPLE 2: Alt statement with guards

 **alt** {

 [v\_x>1] l2.**receive** { // Boolean guard/expression

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

 }

 [v\_x<=1] l2.**receive** { // Boolean guard/expression

 **setverdict**(**inconc**);

 }

 }

EXAMPLE 3: Alt statement with else branch

 // Use of alternative with Boolean expressions (or guard) and else branch

 **alt** {

 :

 [**else**] { // else branch

 f\_myErrorHandling();

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

 **stop;**

 }

 }

EXAMPLE 4: Re-evaluation with repeat

 **alt** {

 [] pCO3.**receive** {

 v\_count := v\_count + 1;

 **repeat** // usage of repeat

 }

 [] t\_t1.**timeout** { }

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

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

 **stop**;

 }

 }

EXAMPLE 5: Alt statement with explicitly invoked altstep

 **alt** {

 [] pCO3.**receive** { }

 [] a\_anotherAltStep() { // Explicit call of altstep a\_anotherAltStep as alternative.

 **setverdict**(**inconc**) // Statement block executed if an alternative within

 // altstep AnotherAltStep has been selected and executed.

 }

 [] t\_myTimer.**timeout** { }

 }

EXAMPLE 6: Alt statement with forbidden function calls

 **alt** {

 [] f\_getPort().**receive**(t(p())) { } // forbidden if f\_getPort, t or p has side effects

 [] a\_anotherAltStep(f()); // forbidden if f has side effects

 [] t\_myTimer[i(p())].**timeout** { } // forbidden if i or p has side effects

 [f\_g()] f\_getComponent(p()).**done** {} // forbidden if f\_g, f\_getComponent or p has side effects

 }

### 20.5.1 The default mechanism

The default mechanism is evoked at the end of each **alt** statement not annotated with the @nodefault modifier, if due to the actual snapshot none of the specified alternatives could be executed. An evoked default mechanism invokes the first altstep in the list of defaults, i.e. the last activated default, and waits for the result of its termination. The termination can be successful or unsuccessful. Unsuccessful means that none of the top alternatives of the **altstep** (see clause 16.1.5) defining the default behaviour could be selected, successful means that one of the top alternatives of the default has been selected and executed.

NOTE 1: An **interleave** statement is semantically equivalent to a nested set of **alt** statements and the default mechanism also applies to each of these **alt** statements. This means, the default mechanism also applies to **interleave** statements. Furthermore, the restrictions imposed on interleave statements in clause 20.4 do not apply to altsteps that are activated as default behaviour for interleave statements.

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

In the case of an unsuccessful termination, the default mechanism invokes the next default in the list. If the last default in the list has terminated unsuccessfully, the default mechanism will return to the place in the **alt** statement in which it has been invoked, i.e. at the end of the **alt** statement, and indicate an unsuccessful default execution. An unsuccessful default execution will also be indicated if the list of defaults is empty.

An unsuccessful default execution may cause a new snapshot or a dynamic error if the test component is blocked (see clause 20.1).

In the case of a successful termination, the default may either stop the test component by means of a **stop** statement, or the main control flow of the test component will continue immediately after the **alt** statement from which the default mechanism was called or the test component will take new snapshot and re-evaluate the **alt** statement. The latter has to be specified by means of a **repeat** statement (see clause 20.3). If the execution of the selected top alternative of the default ends with a **break** statement or without a **repeat** statement the control flow of the test component will continue immediately after the **alt** statement.

NOTE 3: TTCN‑3 does not restrict the implementation of the default mechanism. It may for example be implemented in form of a process that is implicitly called at the end of each **alt** statement or in form of a separate thread that is only responsible for the default handling. The only requirement is that defaults are called in the reverse order of their activation when the default mechanism has been invoked.

### 21.3.7 The Done operation

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

***Syntactical Structure***

( *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 PTC, it matches only if the behaviour of that PTC has been stopped (implicitly or explicitly) or the PTC has been killed. Otherwise, the match is unsuccessful.

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

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

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

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

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

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

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

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clauses 5 and 21 and shown in table 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.

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

***Syntactical Structure***

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

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

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

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

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

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

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clauses 5 and 21 and shown in table 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.

***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**

 }

 }

#### 22.1.4.2 General format of the receiving operations

A receiving operation consists of a *receive* part and an (optional) *assignment* part.

The receive part:

a) specifies the port at which the operation shall take place;

b) defines a matching part which specifies the acceptable input which will match the statement;

c) gives an (optional) address expression that uniquely identifies the communication partner (in case of one‑to‑many connections).

The port name, operation name and value part of all receiving operations shall be present. The identification of the communication partner (denoted by the **from** keyword) is optional and need only be specified in cases of one‑to‑many connections where the receiving entity needs to be explicitly identified.

The assignment part in a receiving operation is optional. For message-based ports it is used when it is required to store received messages. In the case of procedure-based ports it is used for storing the **in** and **inout** parameters of an accepted call, for storing the return value or for storing exceptions. For the message or parameter value assignment part strong typing is not required, e.g. the variable used for storing a message shall be type-compatible to the type of the incoming message.

In addition, the assignment part may also be used to assign the **sender** address of a message, exception, **reply** or **call** to a variable. This is useful for one-to-many connections where, for example, the same message or call can be received from different components, but the message, reply or exception shall be sent back to the original sending component.

For receiving operations using the any port from a port array construction (see clause 22.2.2), the assignment part may also be used to store the indices that identify the specific port instance where the receiving operation matched.

If a receiving operation is used as a stand-alone statement, the **@nodefault** modifier can be placed before it to indicate that the implicit alt statement containing the operation as an alternative shall have the **@nodefault** modifier.

EXAMPLE:

|  |  |  |
| --- | --- | --- |
| Receive part |  | (Optional) assignment part |
| Port and operation | Matching part | (Optional) address expression |  | (Optional) value assignment | (Optional) parameter value assignment | (Optional) sender value assignment |
| myP1.**getreply** | (AProc:{?} **value** 5) |  | -> |  | **param** (v\_v1) | **sender** v\_aPeer |

|  |  |  |
| --- | --- | --- |
| Receive part |  | (Optional) assignment part |
| Port and operation | Matching part | (Optional) address expression |  | (Optional) value assignment | (Optional) parameter value assignment | (Optional) sender value assignment |
| myP2.**receive** | (mw\_myTemplate(5,7)) | **from** v\_aPeer | -> | **value** v\_myVar |  |  |

|  |  |  |
| --- | --- | --- |
| Receive part |  | (Optional) assignment part |
| Port and operation | Matching part | (Optional) address expression |  | (Optional) value assignment | (Optional) parameter value assignment | (Optional) sender value assignment | (Optional)port index assignment |
| **any** **from** p.**receive** | (mw\_myTemplate(5,7)) |  | -> |  |  |  | **@index** **value** v\_i |

## 23.6 The Timeout operation

The **timeout** operation allows to check the expiration of timers.

***Syntactical Structure***

( *ObjectReference* | **any** **timer** | **any from** TimerArrayRef ) "." **timeout**

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

***Semantic Description***

The **timeout** operation allows to check the expiration of a specific timer in the scope unit of a test component or control component in which the timeout operation has been called or of any timer that has been started on a test component or control component before entering the scope in which the **timeout** operation has been called.

When a **timeout** operation is processed, if a timer name is indicated, the timeout-list is searched according to the TTCN‑3 scope rules. If there is a timeout event matching the timer name, that event is removed from the timeout-list, and the **timeout** operation succeeds.

The **timeout** can be used to determine an alternative in an **alt** statement or as stand-alone statement in a behaviour description. In the latter case a **timeout** operation is considered to be shorthand for an **alt** statement with the **timeout** operation as the only alternative. If the **@nodefault** modifier is placed before a stand-alone **timeout** operation, the implicit **alt** statement also contains the **@nodefault** modifier.

The **any** keyword used with the **timeout** operation succeeds if the timeout-list is not empty. In this case a randomly chosen timeout event is removed from the timeout-list.

When the **any from** TimerArrayRefnotation is used, where TimerArrayRef shall be a timer array identifier, the timers from the referenced array are iterated over and individually checked for timeout from innermost to outermost dimension from lowest to highest index for each dimension. The first timer to be found in the timeout-list causes that timer to be removed from the list and the timeout operation succeeds. The index of the matched timer can be optionally stored in an integer variable for single-dimensional arrays or to an integer array or record of integer variable for multi-dimensional timer arrays.

***Restrictions***

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

1. The **timeout** operation does not return any value and therefore shall not be used in an expression.
2. *TimerArrayRef* shall be a reference to a completely initialized timer array.
3. The index redirection shall only be used for **any from** timer array timeout operations.
4. If the index redirection is used for single-dimensional timer arrays, the type of the integer variable shall allow storing the highest index of the respective timer array.
5. If the index redirection is used for multi-dimensional timer arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective timer array, and its type shall allow storing the highest index (from all dimensions) of the timer array.
6. The *ObjectReference* shall be of the timer type.

***Examples***

EXAMPLE 1: Timeout of a specific timer

 t\_myTimer1.**timeout**; // checks for the timeout of the previously started timer MyTimer1

EXAMPLE 2: Timeout of an arbitrary timer

 **any timer.timeout**; // checks for the timeout of any previously started timer

EXAMPLE 3: Timeout of a timer from a timer array

 **timer** t\_myTimerArray[2][2];

 **var integer** v\_i[2];

 **any from** t\_myTimerArray**.timeout -> @index value** v\_i;

 // checks for the timeout of any timer from array

 // assigns index of matched timer to v\_i