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***

[ @nodefault ] *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***

[ @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 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.
10. The **@nodefault** modifier is allowed only in stand-alone **done** statements.

***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***

[ @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 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.
10. The **@nodefault** modifier is allowed only in stand-alone **killed** statements.

***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 |

### 22.2.2 The Receive operation

The **receive** operation is used to receive a message from an incoming message port queue.

***Syntactical Structure***

[ @nodefault ] ( *ObjectReference* | **any** **port** | **any from** PortArrayRef ) "." **receive**

[ "(" *TemplateInstance* ")" ]

[ **from** *Address* ]

[ "->" [ **value** ( ValueRef|

 ( "(" { ValueRef[ ":=" [ **@decoded** [ "(" *Expression* ")" ] ]

 *FieldOrTypeReference* ][","] } ")" )

 ) ]

 [ **sender** ValueRef]

 [ **@index** **value** ValueRef] ]

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**any component**".

***Semantic Description***

The **receive** operation is used to receive a message from an incoming message port queue. The message may be specified by referencing a defined template or can be defined as an in-line template.

The **receive** operation removes the top message from the associated incoming port queue if, and only if, that top message satisfies all the matching criteria associated with the **receive** operation.

If the match is not successful, the top message shall not be removed from the port queue i.e. if the **receive** operation is used as an alternative of an **alt** statement and it is not successful, the execution of **alt** statement shall continue with its next alternative.

**Matching criteria**

The matching criteria are related to the type and value of the message to be received. The type and value of the message to be received are determined by the argument of the **receive** operation, i.e. may either be derived from the defined template or be specified in-line. An optional type field in the matching criteria to the **receive** operation shall be used to avoid any ambiguity of the type of the value being received.

NOTE 2: Encoding attributes also participate in matching in an implicit way, by preventing the decoder to produce an abstract value from the received message encoded in a different way than specified by the attributes.

**Receiving from a specific sender**

In the case of one-to-many connections the **receive** operation may be restricted to a certain communication partner. This restriction shall be denoted using the **from** keyword followed by a specification of an address or component reference, a list of address or component references or **any component**.

NOTE 3: The one-to-one connection is considered to be a simple case of the one-to-many connections and allows the usage of the **from**-clause.

**Storing the received message and parts of the received message**

If the match is successful, the value is removed from the port queue and/or parts of this value can be stored in variables or formal parameters. This is denoted by the symbol '->' and the keyword **value**.

When the keyword **value** is followed by a name of a variable or formal parameter, the whole received message shall be stored in the variable or formal parameter. The variable or formal parameter shall be type compatible with the received message.

When the keyword **value** is followed by a list enframed by a pair of parentheses, the whole received message and/or one or more parts of it can be stored. For each list element that consists only of a variable or formal parameter name the whole message shall be stored in that variable or formal parameter. The type of the variable or formal parameter shall be compatible with the type of the message. Each assignment notation member of the list allows storing the value of the field or element of the received message, which is referenced on the right hand side of the assignment notation (:=), in the variable or formal parameter on the left hand side. The variable or formal parameter shall be type compatible with the type of the referenced field or element.

When assigning individual fields of a message, encoded payload fields can be decoded prior to assignment using the **@decoded** modifier. In this case, the referenced field on the right hand side of the assignment shall be one of the **bitstring**, **hexstring**, **octetstring**, **charstring** or **universal** **charstring** types. It shall be decoded into a value of the same type as the variable on the left hand side of the assignment. Failure of this decoding shall cause a test case error. In case the referenced field is of the **universal** **charstring** type, the **@decoded** clause can contain an optional parameter defining the encoding format. The parameter shall be of the **charstring** type and it shall contain one of the strings allowed for the **decvalue\_unichar** function (specified in clause C.5.4). Any other value shall cause an error. In case the referenced field is not a **universal** **charstring**, the optional parameter shall not be present.

NOTE 4: The model of the behaviour of this implicit decoding is defined in clause B.1.2.9.

NOTE 5: The **@decoded** clause is typically used together with the **decmatch** matching mechanism in the matching part of the receive statement. Since the decoding procedures for assignment and matching are virtually the same, TTCN-3 tools can be optimized in such a way that only one call to the decoder is made when the receiving statement contains both **decmatch** matching mechanism and **@decoded** assignment for the same payload field.

**Storing the sender**

It is also possible to retrieve and store the component reference or address of the sender of a message. This is denoted by the keyword **sender**.

When the message is received on a connected port, only the component reference is stored in the following the **sender** keyword, but the test system shall internally store the component name too, if any (to be used in logging).

**Receive any message**

A **receive** operation with no argument list for the type and value matching criteria of the message to be received shall remove the message on the top of the incoming port queue (if any) if all other matching criteria are fulfilled.

**Receive on any port**

To **receive** a message on any port, use the **any port** keywords.

**Receive on any port from a port array**

To **receive** a message on any port from a specific port array, use the **any from** *PortArrayRef*syntax where PortArrayRefshallbe areference to a port array identifier**.** It is also possible to store the index of a port in a single-dimensional port array at which the operation was successful to a variable of type integer or, in case of multi‑dimensional port arrays the index of the successful port to an integer array or record of integer variable. When checking the port array for matching messages, the port indices to be checked are iterated from lowest to highest. If the port array is multi-dimensional, then the ports are iterated over from innermost to outermost array dimension from lowest to highest index for each dimension, e.g. [0][0], [0][1], [1][0], [1][1]. The first port which matches all the criteria will cause the operation to be successful even if other ports in the array would also meet the criteria.

**Stand-alone receive**

The **receive** operation can be used as a stand-alone statement in a behaviour description. In this latter case the **receive** operation is considered to be shorthand for an **alt** statement with the **receive** operation as the only alternative.

***Restrictions***

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

a) When defining the message in-line, the optional type part shall be present whenever the type of the message being received is ambiguous.

b) The **receive** operation shall only be used on message-based ports and the type of the value to be received shall be included in the list of incoming types of the port type definition.

c) No binding of the incoming values to the terms of the expression or to the template shall occur.

d) A message received by *receive any message* shall not be stored, i.e. the **value** clause shall not be present.

e) Type mismatch at storing the received value or parts of the received value and storing the sender shall cause an error.

NOTE 6: An error due to a type mismatch may happen if the types in the receive part are not compatible to the types in the assignment part or, if the **from** clause is missing, but the type of the sender can be determined and it is not type compatible with the type in the **sender** clause.

f) All *AddressRef* items in the **from** clause and all ValueRef items in the **sender** clause shall be of type **address**, **component** or of the address type bound to the port type (see clause C.5) of the port instance referenced in the **receive** operation. No *AddressRef* in the **from** clause shall contain the special value **null** at the time of the operation.

g) The *PortArrayRef* shall be a reference to a completely initialized port array.

h) The index redirection shall only be used when the operation is used on an any from port array construct.

i) If the index redirection is used for single-dimensional port arrays, the type of the integer variable shall allow storing the highest index of the respective array.

j) If the index redirection is used for multi-dimensional port arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective array, and its type shall allow storing the highest index (from all dimensions) of the array.

k) If a variable referenced in the **value**, **sender** or **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **receive** operation i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **receive** operation.

l) If the **receive** operation contains both **from** and **sender** clause, the variable or parameter referenced in the **sender** clause shall be type compatible with the template in the **from** clause.

m) When assigning implicitly decoded message fields (by using the **@decoded** modifier) in cases where the value or template to be matched uses the *MatchDecodedContent* (**decmatch**) matching for the field to be stored, the type of the template in the *MatchDecodedContent* matching shall be type-compatible to the type of the variable the decoded field is stored into.

n) The referenced value associated with *ValueRef* or the return type associated with *FunctionInstance* followed by the **receive** keyword, shall be of a port type.

o) The **@nodefault** modifier is allowed only in stand-alone **receive** statements.

***Examples***

EXAMPLE 1: Basic receive

 myPort.**receive**(mw\_myTemplate(5, v\_myVar)); // Matches a message that fulfils the conditions

 // defined by template mw\_myTemplate at port myPort.

 myPort.**receive**(v\_a<v\_b); // Matches a Boolean value that depends on the outcome of v\_a<v\_b

 myPort.**receive**(**integer**:v\_myVar); // Matches an integer value with the value of v\_myVar

 // at port myPort

 myPort.**receive**(v\_myVar); // Is an alternative to the previous example

EXAMPLE 2: Receiving from a sender, storing the message, parts of the message or the sender

 **type** MyPayloadType **record** {

 **integer** messageId,

 ContentType content

 }

 **type** MyType2 **record** {

 Header header,

 **octetstring** payload

 }

 **template** MyType mw\_myTemplate := {

 messageId := 42,

 content := ?

 }

 ...

 **var** MyPayloadType v\_myVar;

 **var** **integer** v\_myMessageIdVar, v\_myIntegerVar;

 **var** **charstring** v\_myCharstringVar;

 **var** **address** v\_myPeer;

 **var** **octetstring** v\_myVarOne := '00ff'O;

 MyPort.**receive**(**charstring**:"Hello")**from** v\_myPeer; // Matches charstring "Hello" from MyPeer

 MyPort.**receive**(MyType:?) -> **value** v\_myVar; // The value of the received message is

 // assigned to v\_myVar.

 MyPort.**receive**(MyType:?) -> **value** (v\_myVar, v\_myMessageIdVar:= messageId)

 // The value of the received message is stored in the variable

 // v\_myVar and the value of the messageId field of the received

 // message is stored in the variable v\_myMessageIdVar.

 MyPort.**receive**(anytype:?) -> **value** (v\_myIntegerVar:= integer)

 // If the received value is an integer, it is stored in the variable

 // v\_myIntegerVar, a test case error otherwise.

 MyPort.**receive**(charstring:?) -> **value** (v\_myCharstringVar)

 // The received value is stored in the variable v\_myCharstringVar;

 // Note that it is the same as to write "value v\_myCharstringVar"

 MyPort.**receive**(A<B) -> **sender** v\_myPeer; // The address of the sender is assigned to v\_myPeer

 MyPort.**receive**(MyType:{5, v\_myVarOne }) -> **value** v\_myVar **sender** v\_myPeer;

 // The received message value is stored in v\_myVar and the sender address is stored in

 // v\_myPeer.

 MyPort.**receive**(MyType2:{header := ?, payload := **decmatch** mw\_myTemplate}) -> **value** (v\_myVar := **@decoded** payload);

 // The encoded payload field of the received message is decoded and matched with

 // mw\_myTemplate; if the matching is successful the decoded payload is stored in v\_myVar.

EXAMPLE 3: Receive any message

 myPort.**receive**; // Removes the top value from myPort.

 myPort.**receive** **from** myPeer; // Removes the top message from myPort if its sender is
 // myPeer

 myPort.**receive** -> **sender** v\_mySenderVar; // Removes the top message from myPort and assigns

 // the sender address to v\_mySenderVar

EXAMPLE 4: Receive on any port

 **any port**.**receive**(mw\_myMessage);

EXAMPLE 5: Receive on any port from a port array

 **type** **port** MyPort **message** { **inout** **integer** }

 **type** **component** MyComponent {

 **port** MyPort p[10][10];

 }

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

 **any** **from** p.**receive**(mw\_myMessage) -> **@index value** v\_i;

 // checking receiving mw\_myMessage on any port of the port array p and storing the index of the

 // port on which the matching was successful first; if, for example MyMessage is matched first

 // on p[4,2], the content of i will be {4,2}

### 22.2.3 The Trigger operation

The **trigger** operation is used to await a specific message on an incoming port queue.

***Syntactical Structure***

[ @nodefault ] ( *ObjectReference* | **any** **port** | **any from** PortArrayRef ) "." **trigger**

[ "(" *TemplateInstance* ")" ]

[ **from** *Address* ]

[ "->" [ **value** ( ValueRef|

 ( "(" { ValueRef[ ":=" [ **@decoded** [ "(" *Expression* ")" ] ]

 *FieldOrTypeReference* ][","] } ")" )

 ) ]

 [ **sender** ValueRef]

 [ **@index** **value** ValueRef] ]

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**any component**".

***Semantic Description***

The **trigger** operation removes the top message from the associated incoming port queue. If that top message meets the matching criteria, the **trigger** operation behaves in the same manner as a **receive** operation. If that top message does not fulfil the matching criteria, it shall be removed from the queue without any further action.

The **trigger** operation requires the port name, matching criteria for type and value, an optional **from** restriction (i.e. selection of communication partner) and an optional assignment of the matching message and sender component to variables.

**Matching criteria**

The matching criteria as defined in clause 22.2.2 apply also to the **trigger** operation.

**Trigger from a specific sender**

In the case of one-to-many connections the **trigger** operation may be restricted to a certain communication partner. This restriction shall be denoted using the **from** keyword followed by a specification of an address or component reference, a list of address or component references or **any component**.

NOTE 2: The one-to-one connection is considered to be a simple case of the one-to-many connections and allows the usage of the **from**-clause.

**Trigger on any message**

A **trigger** operation with no argument list shall trigger on the receipt of any message. Thus, its meaning is identical to the meaning of receive any message.

**Trigger on any port**

To **trigger** on a message at any port, use the **any port** keywords.

**Trigger on any port from a port array**

To trigger on a message at any port from a specific port array, use the **any from** *PortArrayRef*syntax where PortArrayRefshallbe areference to a port array identifier**.** It is also possible to store the index of a port in a single‑dimensional port array at which the operation was successful to a variable of type integer or, in case of multi‑dimensional port arrays the index of the successful port to an integer array or record of integer variable. When checking the port array for matching messages, the port indices to be checked are iterated from lowest to highest. If the port array is multi-dimensional, then the ports are iterated over from innermost to outermost array dimension from lowest to highest index for each dimension, e.g. [0][0], [0][1], [1][0], [1][1]. The first port which matches all the criteria will cause the operation to be successful even if other ports in the array would also meet the criteria.

If any port in the port array which is checked for matching contains a message that does not match, this message is removed and the containing **alt** statement is re-evaluated, regardless of whether or not other ports in the port array would meet the trigger criteria.

**Stand-alone trigger**

The **trigger** operation can be used as a stand-alone statement in a behaviour description. In this latter case the **trigger** operation is considered to be shorthand for an **alt** statement with two alternatives (one alternative expecting the message and another alternative consuming all other messages and repeating the alt statement, see ETSI ES 201 873‑4 [1]).

**Storing the received message, parts of the received message or the sender**

Rules in clause 22.2.2 shall apply.

***Restrictions***

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

a) The **trigger** operation shall only be used on message-based ports and the type of the value to be received shall be included in the list of incoming types of the port type definition.

b) A message received by *TriggerOnAnyMessage* shall not be assigned to a variable.

c) Type mismatch at storing the received value or parts of the received value and storing the sender shall cause an error.

NOTE 3: An error due to a type mismatch may happen if the types in the receive part are not compatible to the types in the assignment part or, if the **from** clause is missing, but the type of the sender can be determined and it is not type compatible with the type in the **sender** clause.

d) All *AddressRef* items in the **from** clause and all ValueRefitems in the **sender** clause shall be of type **address**, **component** or of the address type bound to the port type (see clause C.5) of the port instance referenced in the **trigger** operation. No *AddressRef* in the **from** clause shall contain the special value **null** at the time of the operation.

e) The *PortArrayRef* shall be a reference to a completely initialized port array.

f) The index redirection shall only be used when the operation is used on an any from port array construct.

g) If the index redirection is used for single-dimensional port arrays, the type of the integer variable shall allow storing the highest index of the respective array.

h) If the index redirection is used for multi-dimensional port arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective array, and its type shall allow storing the highest index (from all dimensions) of the array.

i) If a variable referenced in the **value**, **sender** or **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **trigger** operation, i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **trigger** operation.

j) If the **trigger** operation contains both **from** and **sender** clause, the variable or parameter referenced in the **sender** clause shall be type compatible with the template in the **from** clause.

k) The *ObjectReference* shall be of a port type.

l) The **@nodefault** modifier is allowed only in stand-alone **trigger** statements.

***Examples***

EXAMPLE 1: Basic trigger

 myPort.**trigger**(MyType:?);

 // Specifies that the operation will trigger on the reception of the first message observed of

 // the type MyType with an arbitrary value at port myPort.

EXAMPLE 2: Trigger from a sender and with storing message or sender

 myPort.**trigger**(MyType:?) **from** myPartner;

 // Triggers on the reception of the first message of type MyType at port myPort

 // received from myPartner.

 myPort.**trigger**(MyType:?) **from** myPartner -> **value** v\_myRecMessage;

 // This example is almost identical to the previous example. In addition, the message which

 // triggers i.e. all matching criteria are met, is stored in the variable v\_myRecMessage.

 myPort.**trigger**(MyType:?) -> **sender** myPartner;

 // This example is almost identical to the first example. In addition, the reference of the

 // sender component will be retrieved and stored in variable myPartner.

 myPort.**trigger**(**integer**:?) -> **value** v\_myVar **sender** v\_myPartner;

 // Trigger on the reception of an arbitrary integer value which afterwards is stored in

 // variable v\_myVar. The reference of the sender component will be stored in variable MyPartner.

EXAMPLE 3: Trigger on any message

 myPort.**trigger**;

 myPort.**trigger** **from** myPartner;

 myPort.**trigger** -> **sender** v\_mySenderVar;

EXAMPLE 4: Trigger on any port

 **any port**.**trigger**

EXAMPLE 5: Trigger on any port from port array

 **type** **port** MyPort **message** { **inout** **integer** }

 **type** **component** MyComponent {

 **port** MyPort p[10][10];

 }

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

 **any** **from** p.**trigger**(mw\_myMessage) -> **@index** **value** v\_i;

 // Checking if mw\_myMessage has been received on any port of the port array p; if yes, the index

 // of the port on which the matching was first successful is stored in the array v\_i; if no port

 // succeeds, the top messages are removed and the port array is re-checked.

### 22.3.2 The Getcall operation

The **getcall** operation is used to accept calls.

***Syntactical Structure***

[ @nodefault ] ( *ObjectReference* | **any** **port** | **any from** PortArrayRef ) "." **getcall**

[ "(" *TemplateInstance* ")" ]

[ **from** *Address* ]

[ "->" [ **param** "(" { ( ValueRef":=" [ **@decoded** [ "(" *Expression* ")" ] ] *ParameterIdentifier* ) "," } *|*

 { ( ValueRef| "-" ) "," }

 ")" ]

 [ **sender** ValueRef]

 [ @**index** **value** ValueRef] ]

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**any component**".

***Semantic Description***

The **getcall** operation is used to specify that a test component accepts a call from the SUT, or another test component.

The **getcall** operation shall remove the top call from the incoming port queue, if, and only if, the matching criteria associated to the **getcall** operation are fulfilled. These matching criteria are related to the signature of the call to be processed and the communication partner. The matching criteria for the signature may either be specified in-line or be derived from a signature template.

The assignment of **in** and **inout** parameter values to variables shall be made in the assignment part of the **getcall** operation. This allows the use of signature templates in **getcall** operations in the same manner as templates are used for types.

A **getcall** operation may be restricted to a certain communication partner in case of one-to-many connections. This restriction shall be denoted by using the **from** keyword followed by a specification of an address or component reference, a list of address or component references or **any component**.

NOTE 2: The one-to-one connection is considered to be a simple case of the one-to-many connections and allows the usage of the **from**-clause.

The (optional) assignment part of the **getcall** operation comprises the assignment of **in** and **inout** parameter values to variables and the retrieval of the address of the calling component. The keyword **param** is used to retrieve the parameter values of a call.

When assigning individual parameters of a call, encoded parameters can be decoded prior to assignment using the **@decoded** modifier. In this case, the referenced parameter on the right hand sided of the assignment shall be one of the **bitstring**, **hexstring**, **octetstring**, **charstring** or **universal** **charstring** types. It shall be decoded into a value of the same type as the variable on the left hand side of the assignment. Failure of this decoding shall cause a test case error. In case the referenced field is of the **universal** **charstring** type, the **@decoded** clause can contain an optional parameter defining the encoding format. The parameter shall be of the **charstring** type and it shall contain one of the strings allowed for the **decvalue\_unichar** function (specified in clause C.5.4). Any other value shall cause an error. In case the referenced field is not a **universal** **charstring**, the optional parameter shall not be present.

The keyword **sender** is used when it is required to retrieve the address of the sender (e.g. for addressing a **reply** or exception to the calling party in a one-to-many configuration).

**Accepting any call**

A **getcall** operation with no argument list for the signature matching criteria will remove the call on the top of the incoming port queue (if any) if all other matching criteria are fulfilled.

**Getcall on any port**

To **getcall** on any port is denoted by the **any** keyword.

**Getcall on any port from a port array**

To **getcall** on any port from a specific port array, use the **any from** *PortArrayRef*syntax where PortArrayRefshallbe areference to a port array identifier**.** It is also possible to store the index of a port in a single-dimensional port array at which the operation was successful to a variable of type integer or, in case of multi‑dimensional port arrays the index of the successful port to an integer array or record of integer variable. When checking the port array for matching calls, the port indices to be checked are iterated from lowest to highest. If the port array is multi-dimensional, then the ports are iterated over from innermost to outermost array dimension from lowest to highest index for each dimension, e.g. [0][0], [0][1], [1][0], [1][1]. The first port which matches all the criteria will cause the operation to be successful even if other ports in the array would also meet the criteria.

***Restrictions***

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

a) The **getcall** operation shall only be used on procedure-based ports. The type definition of the port shall include the name of the procedure to which the getcall operation belongs in its **in** or **inout** list.

b) The signature argument of the **getcall** operation shall not be used to pass in variable names for **in** and **inout** parameters.

c) The *ParameterIdentifier*s shall be from the corresponding signature definition.

d) The value assignment part shall not be used with the getcall operation.

e) Parameters of calls accepted by *accepting any call* shall not be assigned to a variable, i.e. the **param** clause shall not be present.

f)All *AddressRef* items in the **from** clause and all ValueRef items in the **sender** clause shall be of type **address**, **component** or of the address type bound to the port type (see clause C.5) of the port instance referenced in the **getcall** operation. No *AddressRef* in the **from** clause shall contain the special value **null** at the time of the operation.

g) The *PortArrayRef* shall be a reference to a completely initialized port array.

h) The index redirection shall only be used when the operation is used on an any from port array construct.

i) If the index redirection is used for single-dimensional port arrays, the type of the integer variable shall allow storing the highest index of the respective array.

j) If the index redirection is used for multi-dimensional port arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective array, and its type shall allow storing the highest index (from all dimensions) of the array.

k) If a variable referenced in the **param**, **sender** or **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **getcall** operation, i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **getcall** operation.

l) If the **getcall** operation contains both **from** and **sender** clause, the variable or parameter referenced in the **sender** clause shall be type compatible with the template in the **from** clause. If the operation contains a **sender** clause but no **from** clause, the sender shall be type compatible with the type of the variable or parameter referenced in the **sender** clause.

NOTE 3: An error due to a type mismatch may happen if the types in the receive part are not compatible to the types in the assignment part or, if the **from** clause is missing, but the type of the sender can be determined and it is not type compatible with the type in the **sender** clause.

m) When assigning implicitly decoded parameters (by using the **@decoded** modifier) in cases where the value or template to be matched uses the *MatchDecodedContent* (**decmatch**) matching for the parameter to be stored, the type of the template in the *MatchDecodedContent* matching shall be type-compatible to the type of the variable the decoded field is stored into.

n) The *ObjectReference* shall be of a port type.

o) The **@nodefault** modifier is allowed only in stand-alone **getcall** statements.

***Examples***

EXAMPLE 1: Basic getcall

 myPort.**getcall**(MyProc: s\_myProcTemplate(5, v\_myVar)); // accepts a call of MyProc at myPort

 myPort.**getcall**(MyProc:{5, v\_myVar}) **from** myPeer; // accepts a call of MyProc at myPort from

 // myPeer

EXAMPLE 2: Getcall with matching and assignments of parameter values to variables

 myPort.**getcall**(MyProc:{?, ?}) **from** myPartner -> **param** (v\_myPar1Var, v\_myPar2Var);

 // The in or inout parameter values of MyProc are assigned to v\_myPar1Var and v\_myPar2Var.

 myPort.**getcall**(MyProc:{5, v\_myVar}) -> **sender** v\_mySenderVar;

 // Accepts a call of MyProc at myPort with the in or inout parameters 5 and v\_myVar.

 // The address of the calling party is retrieved and stored in v\_mySenderVar.

 // The following getcall examples show the possibilities to use matching attributes

 // and omit optional parts, which may be of no importance for the test specification.

 myPort.**getcall**(MyProc:{5, v\_myVar}) -> **param**(v\_myVar1, v\_myVar2) **sender** v\_mySenderVar;

 myPort.**getcall**(MyProc:{5, ?}) -> **param**(v\_myVar1, v\_myVar2);

 myPort.**getcall**(MyProc:{?, v\_myVar}) -> **param**( - , v\_myVar2);

 // The value of the first inout parameter is not important or not used

 // The following examples shall explain the possibilities to assign in and inout parameter

 // values to variables. The following signature is assumed for the procedure to be called:

 **signature** MyProc2(**in** **integer** A, **integer** B, **integer** C, **out** **integer** D, **inout** **integer** E);

 myPort.**getcall**(MyProc2:{?, ?, 3, - , ?}) -> **param** (v\_myVarA, v\_myVarB, - , -, v\_myVarE);

 // The parameters A, B, and E are assigned to the variables v\_myVarA, v\_myVarB, and

 // v\_myVarE. The out parameter D needs not to be considered.

 myPort.**getcall**(MyProc2:{?, ?, 3, -, ?}) -> **param** (v\_myVarA:= A, v\_myVarB:= B, v\_myVarE:= E);

 // Alternative notation for the value assignment of in and inout parameter to variables. Note,

 // the names in the assignment list refer to the names used in the signature of MyProc2

 myPort.**getcall**(MyProc2:{1, 2, 3, -, \*}) -> **param** (v\_myVarE:= E);

 // Only the inout parameter value is needed for the further test case execution

 // The following example demonstrates the use of encoded parameters:

 **signature** MyProc3(**in** **integer** paramType, **octetstring** encodedParam);

 **template integer** mw\_int := ?;

 …

 **var integer** v\_myVarX;

 myPort.**getcall**(MyProc3:{1, **decmatch** mw\_int}) -> **param** (v\_myVarX := **@decoded** encodedParam);

 // The parameters encodedParam is decoded into an integer and assigned to v\_myVarX.

EXAMPLE 3: Accepting any call

 myPort.**getcall**; // Removes the top call from myPort.

 myPort.**getcall** **from** myPartner; // Removes a call from myPartner from port myPort

 myPort.**getcall** -> **sender** v\_mySenderVar; // Removes a call from myPort and retrieves

 // the address of the calling entity

EXAMPLE 4: Getcall on any port

 **any** **port**.**getcall**(MyProc:?)

EXAMPLE 5: Getcall on any port from port array

 **type** **port** MyPort **procedure** { **inout** MyProc }

 **type** **component** MyComponent {

 **port** **MyPort** p[10][10];

 }

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

 **any** **from** p.**getcall**(MyProc:?) -> **@index** **value** v\_i;

 // checking for an incoming call of the type MyProc on any port of the port array p and storing

 // the index of the port on which the matching was successful first

### 22.3.4 The Getreply operation

The **getreply** operation is used to handle replies from a previously called procedure.

***Syntactical Structure***

[ @nodefault ] ( *ObjectReference* | **any** **port** | **any from** PortArrayRef ) "." **getreply**

[ "(" *TemplateInstance* [ **value** *TemplateInstance* ]")" ]

[ **from** *Address* ]

[ "->" [ **value** (ValueRef|

 ( "(" { ValueRef[ ":=" [ **@decoded** [ "(" *Expression* ")" ] ]
 *FieldOrTypeReference* ][","] } ")" )

 )]

 [ **param** "(" { ( ValueRef":=" [ **@decoded** [ "(" *Expression* ")" ] ]
 *ParameterIdentifier* ) "," } *|*

 { ( ValueRef| "-" ) "," }

 ")" ]

 [ **sender** ValueRef]

 [ @**index** **value** ValueRef] ]

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**any component**".

***Semantic Description***

The **getreply** operation is used to handle replies from a previously called procedure.

The **getreply** operation shall remove the top reply from the incoming port queue, if, and only if, the matching criteria associated to the **getreply** operation are fulfilled. These matching criteria are related to the signature of the procedure to be processed and the communication partner. The matching criteria for the signature may either be specified in-line or be derived from a signature template.

Matching against a received return value can be specified by using the **value** keyword.

A **getreply** operation may be restricted to a certain communication partner in case of one-to-many connections. This restriction shall be denoted by using the **from** keyword followed by a specification of an address or component reference, a list of address or component references or **any component**.

NOTE 2: The one-to-one connection is considered to be a simple case of the one-to-many connections and allows the usage of the **from**-clause.

The assignment of **out** and **inout** parameter values to variables shall be made in the assignment part of the **getreply** operation. This allows the use of signature templates in **getreply** operations in the same manner as templates are used for types.

The (optional) assignment part of the **getreply** operation comprises the assignment of **out** and **inout** parameter values to variables and the retrieval of the address of the sender of the reply. The keyword **value** is used to retrieve return values and the keyword **param** is used to retrieve the parameter values of a reply. The keyword **sender** is used when it is required to retrieve the address of the sender.

When assigning individual parameters or referenced fields of the return value of a reply, encoded parameters can be decoded prior to assignment using the **@decoded** modifier. In this case, the referenced parameter or field of the return value on the right hand sided of the assignment shall be one of the **bitstring**, **hexstring**, **octetstring**, **charstring** or **universal** **charstring** types. It shall be decoded into a value of the same type as the variable on the left hand side of the assignment. Failure of this decoding shall cause a test case error. In case the parameter or referenced field of the return value is of the **universal** **charstring** type, the **@decoded** clause can contain an optional parameter defining the encoding format. The parameter shall be of the **charstring** type and it shall contain one of the strings allowed for the **decvalue\_unichar** function (specified in clause C.5.4). Any other value shall cause an error. In case the parameter or referenced field of the return value is not a **universal** **char string**, the optional parameter shall not be present.

**Get any reply**

A **getreply** operation with no argument list for the signature matching criteria shall remove the reply message on the top of the incoming port queue (if any) if all other matching criteria are fulfilled.

If *GetAnyReply* is used in the response and exception handling part of a **call** operation, it shall only treat replies from the procedure invoked by the **call** operation.

**Get a reply on any port**

To get a reply on any port, use the **any port** keywords.

**Get a reply on any port from a port array**

To get a reply on any port from a specific port array, use the **any from** *PortArrayRef*syntax where PortArrayRefshallbe areference to a port array identifier**.** It is also possible to store the index of a port in a single‑dimensional port array at which the operation was successful to a variable of type integer or, in case of multi‑dimensional port arrays the index of the successful port to an integer array or record of integer variable. When checking the port array for matching replies, the port indices to be checked are iterated from lowest to highest. If the port array is multi-dimensional, then the ports are iterated over from innermost to outermost array dimension from lowest to highest index for each dimension, e.g. [0][0], [0][1], [1][0], [1][1]. The first port which matches all the criteria will cause the operation to be successful even if other ports in the array would also meet the criteria.

***Restrictions***

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

a) A **getreply** operation shall only be used at a procedure-based port. The type definition of the port shall include the name of the procedure to which the **getreply** operation belongs.

b) The signature argument of the **getreply** operation shall not be used to pass in variable names for **out** and **inout** parameters.

c) Parameters or return values of responses accepted by *get any reply* shall not be assigned to a variable, i.e. the **param** and **value** clause shall not be present.

d)All *AddressRef* items in the **from** clause and all ValueRef items in the **sender** clause shall be of type **address**, **component** or of the address type bound to the port type (see clause 6.2.9) of the port instance referenced in the **getreply** operation. No *AddressRef* in the **from** clause shall contain the special value **null** at the time of the operation.

e) The *PortArrayRef* shall be a reference to a completely initialized port array.

f) The index redirection shall only be used when the operation is used on an any from port array construct.

g) If the index redirection is used for single-dimensional arrays, the type of the integer variable shall allow storing the highest index of the respective port array.

h) If the index redirection is used for multi-dimensional arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective port array, and the its type shall allow storing the highest index (from all dimensions) of the port array.

i) If a variable referenced in the **value**, **param**, **sender** or **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **getreply** operation, i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **getreply** operation.

j) If the **getreply** operation contains both **from** and **sender** clause, the variable or parameter referenced in the **sender** clause shall be type compatible with the template in the **from** clause. If the operation contains a **sender** clause but no **from** clause, the sender shall be type compatible with the variable or parameter referenced in the **sender** clause.

NOTE 3: An error due to a type mismatch may happen if the types in the receive part are not compatible to the types in the assignment part or, if the **from** clause is missing, but the type of the sender can be determined and it is not type compatible with the type in the **sender** clause.

k) When assigning implicitly decoded parameters or referenced fields of the return value (by using the **@decoded** modifier) in cases where the value or template to be matched uses the *MatchDecodedContent* (**decmatch**) matching for the parameter to be stored, the type of the template in the *MatchDecodedContent* matching shall be type-compatible to the type of the variable the decoded field is stored into.

l) The ObjectReference shall be of a port type.

m) The **@nodefault** modifier is allowed only in stand-alone **getreply** statements.

***Examples***

EXAMPLE 1: Basic getreply

 myPort.**getreply**(MyProc:{5, ?} **value** 20); // Accepts a reply of MyProc with two out or

 // inout parameters and a return value of 20

 myPort.**getreply**(MyProc2:{ - ,5}) **from** myPeer; // Accepts a reply of MyProc2 from myPeer

EXAMPLE 2: Getreply with storing inout/out parameters and return values in variables

 myPort.**getreply**(MyProc1:{?, ?} **value** ?) -> **value** v\_myRetValue **param**(v\_myPar1, v\_myPar2);

 // The returned value is assigned to variable v\_myRetValue and the value

 // of the two out or inout parameters are assigned to the variables v\_myPar1 and v\_myPar2.

 myPort.**getreply**(MyProc1:{?, ?} **value** ?)-> **value** v\_myRetValue **param**(- ,v\_myPar2) **sender** mySender;

 // The value of the first parameter is not considered for the further test execution and

 // the address of the sender component is retrieved and stored in the variable mySender.

 // The following examples describe some possibilities to assign out and inout parameter values

 // to variables. The following signature is assumed for the procedure which has been called

 **signature** MyProc2(**in** **integer** A, **integer** B, **integer** C, **out** **integer** D, **inout** **integer** E);

 myPort.**getreply**(s\_aTemplate) -> **param**( - , - , - , v\_myVarOut1, v\_myVarInout1);

 myPort.**getreply**(s\_aTemplate) -> **param**(v\_myVarOut1:=D, v\_myVarOut2:=E);

 myPort.**getreply**(MyProc2:{ - , - , - , 3, ?}) -> **param**(v\_myVarInout1:=E);

 // The following example demonstrates the use of encoded parameters:

 **signature** MyProc3(**out** **integer** paramType, **out** **octetstring** encodedParam);

 **template integer** mw\_int := ?;

 …

 **var integer** v\_myVarX;

 myPort.**getreply**(MyProc3:{1, **decmatch** mw\_int}) -> **param** (v\_myVarX := **@decoded** encodedParam);

 // The parameters encodedParam is decoded into an integer and assigned to v\_myVarX.

EXAMPLE 3: Get any reply

 myPort.**getreply**; // Removes the top reply from myPort.

 myPort.**getreply** **from** myPeer; // Removes the top reply received from myPeer from myPort.

 myPort.**getreply** -> **sender** v\_mySenderVar; // Removes the top reply from myPort and retrieves

 // the address of the sender entity

EXAMPLE 4: Get a reply on any port

 **any** **port**.**getreply**(Myproc:?)

EXAMPLE 5: Get a reply on any port from port array

 **type** **port** MyPort **procedure** { **inout** MyProc }

 **type** **component** MyComponent {

 **port** MyPort p[10][10];

 }

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

 **any** **from** p.**getreply**(MyProc:?) -> **@index** **value** v\_i;

 // Getting a reply of the type MyProc on any port of the port array p and

 // storing the index of the port on which the matching was successful first

### 22.3.6 The Catch operation

The **catch** operation is used to catch exceptions.

***Syntactical Structure***

[ @nodefault ] ( *ObjectReference* | **any** **port** | **any from** PortArrayRef ) "." **catch**

[ "(" ( *Signature* [ "," *TemplateInstance* ] ) | *TimeoutKeyword* ")" ]

[ **from** *Address* ]

[ "->" [ **value** ( ValueRef|

 ( "(" { ValueRef[ ":=" [ **@decoded** [ "(" *Expression* ")" ] ] *FieldOrTypeReference* ][","] } ")" )

 ) ]

 [ **sender** ValueRef]

 [ **@index** **value** ValueRef] ]

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**any component**".

***Semantic Description***

The **catch** operation is used to catch exceptions raised by a test component or the SUT as a reaction to a procedure call. Exceptions are specified as types and thus, can be treated like messages, e.g. templates can be used to distinguish between different values of the same exception type. If a *Signature* is given in the parameter list, it is possible to omit the *TemplateInstance* part if the **catch** operation shall match any exception value of any of the exception types declared in the definition of the referenced *Signature*.

The **catch** operation removes the top exception from the associated incoming port queue if, and only if, that top exception satisfies all the matching criteria associated with the **catch** operation.

A **catch** operation may be restricted to a certain communication partner in case of one-to-many connections. This restriction shall be denoted by using the **from** keyword followed by a specification of an address or component reference, a list of address or component references or **any component**.

NOTE 2: The one-to-one connection is considered to be a simple case of the one-to-many connections and allows the usage of the **from**-clause.

The (optional) redirection part of the **catch** operation comprises of storing the exception value and/or one or more parts of it and the retrieval of the address of the calling component. The keyword **value** is used to retrieve the value of an exception and/or the parts of it and the keyword **sender** is used when it is required to retrieve the address of the sender.

When assigning individual fields of an exception, encoded payload fields can be decoded prior to assignment using the **@decoded** modifier. In this case, the referenced field on the right hand sided of the assignment shall be one of the **bitstring**, **hexstring**, **octetstring**, **charstring** or **universal** **charstring** types. It shall be decoded into a value of the same type as the variable on the left hand side of the assignment. Failure of this decoding shall cause a test case error. In case the referenced field is of the **universal** **charstring** type, the **@decoded** clause can contain an optional parameter defining the encoding format. The parameter shall be of the **charstring** type and it shall contain one of the strings allowed for the **decvalue\_unichar** function (specified in clause C.5.4). Any other value shall cause an error. In case the referenced field is not a **universal** **charstring**, the optional parameter shall not be present.

The **catch** operation may be part of the response and exception handling part of a **call** operation or be used to determine an alternative in an **alt** statement. If the **catch** operation is used in the accepting part of a **call** operation, the information about port name and signature reference to indicate the procedure that raised the exception is redundant, because this information follows from the **call** operation. However, for readability reasons (e.g. in case of complex **call** statements) this information shall be repeated.

**The Timeout exception**

There is one special **timeout** exception that can be caught by the **catch** operation. The **timeout** exception is an emergency exit for cases where a called procedure neither replies nor raises an exception within a predetermined time (see clause 22.3.1).

**Catch any exception**

A **catch** operation with no argument list allows any valid exception to be caught. The most general case is without using the **from** keyword. *CatchAnyException* will also catch the **timeout** exception.

**Catch any exception for specific signature**

A **catch** operation using only a *Signature* reference in the argument list allows any valid exception for that signature to be caught.

**Catch on any port**

To **catch** an exception on any port use the **any** keyword.

**Catch on any port from a port array**

To **catch** an exception on any port from a specific port array, indices use the **any from** *PortArrayRef*syntax where PortArrayRefshallbe areference to a port array identifier**.** It is also possible to store the index of a port in a single-dimensional port array at which the operation was successful to a variable of type integer or, in case of multi‑dimensional port arrays the index of the successful port to an integer array or record of integer variable. When checking the port array for matching exceptions, the port indices to be checked are iterated from lowest to highest. If the port array is multi-dimensional, then the ports are iterated over from innermost to outermost array dimension from lowest to highest index for each dimension, e.g. [0][0], [0][1], [1][0], [1][1]. The first port which matches all the criteria will cause the operation to be successful even if other ports in the array would also meet the criteria.

The catch on any port from a port array operation cannot be used to catch a call timeout.

***Restrictions***

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

1. The catch operation shall only be used at procedure-based ports. The type of the caught exception shall be specified in the signature of the procedure that raised the exception.
2. The type definition of the port shall include in its out or inout list the name of the procedure to which the exception belongs.
3. No binding of the incoming values to the terms of the expression or to the template shall occur. The assignment of the exception values to variables shall be made in the assignment part of the catch operation.
4. Catching timeout exceptions shall be restricted to the exception handling part of a call. No further matching criteria (including a from part) and no assignment part is allowed for a catch operation that handles a timeout exception.
5. Exception values accepted by catch any exception shall not be assigned to a variable, i.e. the value clause shall not be present.
6. If CatchAnyException is used in the response and exception handling part of a call operation, it shall only treat exceptions raised by the procedure invoked by the call operation.
7. All AddressRef items in the from clause and all ValueRef items in the sender clause shall be of type address, component or of the address type bound to the port type (see clause 6.2.9) of the port instance referenced in the catch operation. No AddressRef in the from clause shall contain the special value null at the time of the operation.
8. The PortArrayRef shall be a reference to a completely initialized port array.
9. The index redirection shall only be used when the operation is used on an any from port array construct.
10. If the index redirection is used for single-dimensional arrays, the type of the integer variable shall allow storing the highest index of the respective port array.
11. If the index redirection is used for multi-dimensional arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective port array, and the its type shall allow storing the highest index (from all dimensions) of the port array.
12. If a variable referenced in the value, sender or @index clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the catch operation, i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the catch operation.
13. If the catch operation contains both from and sender clause, the variable or parameter referenced in the sender clause shall be type compatible with the template in the from clause. If the operation contains a sender clause but no from clause, the sender shall be type compatible with the variable or parameter referenced in the sender clause.

NOTE 3: An error due to a type mismatch may happen if the types in the receive part are not compatible to the types in the assignment part or, if the from clause is missing, but the type of the sender can be determined and it is not type compatible with the type in the sender clause.

1. When assigning implicitly decoded exception fields (by using the @decoded modifier) in cases where the value or template to be matched uses the MatchDecodedContent (decmatch) matching for the parameter to be stored, the type of the template in the MatchDecodedContent matching shall be type-compatible to the type of the variable the decoded field is stored into.
2. The referenced value associated with Ref or the return type associated with FunctionInstance followed by the catch keyword, shall be of a port type.
3. If no *TemplateInstance* is provided in the parameter list, then also no **value** clause shall be present in the redirection part.
4. The **@nodefault** modifier is allowed only in stand-alone **catch** statements.

***Examples***

EXAMPLE 1: Basic catch

 myPort.**catch**(MyProc, **integer:** v\_myVar); // Catches an integer exception of value

 // v\_myVar raised by MyProc at port myPort.

 myPort.**catch**(MyProc, v\_myVar); // Is an alternative to the previous example.

 myPort.**catch**(MyProc, v\_a<v\_b); // Catches a boolean exception

 myPort.**catch**(MyProc, MyType:{5, v\_myVar}); // In-line template definition of an exception value.

 myPort.**catch**(MyProc, **charstring**:"Hello")**from** myPeer; // Catches "Hello" exception from myPeer

EXAMPLE 2: Catch with storing value and/or sender in variables

 myPort.**catch**(MyProc, MyType:?) **from** myPartner -> **value** v\_myVar;

 // Catches an exception from myPartner and assigns its value to v\_myVar.

 myPort.**catch**(MyProc, s\_myTemplate(5)) -> **value** v\_myVarTwo **sender** myPeer;

 // Catches an exception, assigns its value to v\_myVarTwo and retrieves the

 // address of the sender.

 myPort.**catch**(MyProc, s\_myTemplate(5)) -> **value** (v\_myVarThree:= f1)

 **sender** myPeer;

 // Catches an exception, assigns the value of its field f1 to v\_myVarThree and retrieves the

 // address of the sender.

 // Handling encoded exception payload:

 **type** MyException **record** {

 **...**

 }

 **type** CommonException **record** {

 **integer** exceptionId,

 **octetstring** payload

 }

 **signature** S() **exception** (CommonException);

 ...

 **var** MyException v\_myVar;

 myPort.**catch** (S, CommonException:{exceptionId := 25, payload := **decmatch** MyException:? }) -> **value** (v\_myVar := **@decoded** payload);

 // The encoded payload field of the caught exception is decoded and matched with m\_excTemplate;

 // if the matching is successful the decoded payload is stored in v\_myVar.

EXAMPLE 3: The Timeout exception

 myPort.**call**(MyProc:{5, v\_myVar}, 20E-3) {

 [] myPort.**getreply**(MyProc:{?, ?}) { }

 [] myPort.**catch**(**timeout**) { // timeout exception after 20ms

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

 **stop**;

 }

 }

EXAMPLE 4: Catch any exception

 myPort.**catch**;

 myPort.**catch** **from** myPartner;

 myPort.**catch** -> **sender** v\_mySenderVar;

 myPort.catch(MyProc); // catch any exception raised by procedure MyProc

EXAMPLE 5: Catch on any port

 **any port**.**catch;**

EXAMPLE 6: Catch on any port from port array

 **type** **port** MyPort **procedure** { **inout** MyProc }

 **type** **component** MyComponent {

 **port** MyPort p[10][10];

 }

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

 **any** **from** p.**catch**(MyProc, MyType:?) -> **@index** **value** v\_i;

 // Catching an incoming exception of type MyType on any port in the port array p and

 // storing the index of the port on which the matching was successful first

## 22.4 The Check operation

The **check** operation allows reading the top element of a message‑based or procedure‑based *incoming* port queue.

***Syntactical Structure***

[ @nodefault ] ( *ObjectReference* | **any** **port** | **any from** PortArrayRef ) "." **check**

[ "("

 ( *PortReceiveOp | PortGetCallOp | PortGetReplyOp | PortCatchOp* ) |

 ( [ **from** *Address* ]

 [ "->" [ **sender** ValueRef]

 [ **@index** **value** ValueRef] ] )

 ")" ]

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**any component**".

***Semantic Description***

The **check** operation is a generic operation that allows read access to the top element of message‑based and procedure‑based *incoming* port queues without removing the top element from the queue. The **check** operation has to handle values of a certain type at message-based ports and to distinguish between calls to be accepted, exceptions to be caught and replies from previous calls at procedure-based ports.

The receiving operations **receive**, **getcall**, **getreply** and **catch** together with their matching and value, sender or parameter storing parts, are used by the **check** operation to define the conditions that have to be checked and the information to be optionally extracted.

It is the *top* element of an incoming port queue that shall be checked (it is not possible to look *into* the queue). If the queue is empty the **check** operation fails. If the queue is not empty, a copy of the top element is taken and the receiving operation specified in the **check** operation is performed on the copy. The **check** operation fails if the receiving operation fails i.e. the matching criteria are not fulfilled. In this case the *copy* of the top element of the queue is discarded and test execution continues in the normal manner, i.e. the statement or alternative next to the check operation is evaluated. The **check** operation is successful if the receiving operation is successful. In this case, the value, sender or parameter storing parts of the receiving operation, if any, are executed, i.e. the message and/or a part of it, the sender's address or component reference, the parameter(s) of the call or reply or the value of the exception are stored in the associated variables.

If **check** is used as a stand-alone statement, it is considered to be a shorthand for an **alt** statement with the **check** operation as the only alternative.

**Check from a specific sender**

In the case of one-to-many connections the **check** operation may be restricted to a certain communication partner. This restriction shall be denoted using the **from** keyword followed by a specification of an address or component reference, a list of address or component references or **any component**.

NOTE 2: The one-to-one connection is considered to be a simple case of the one-to-many connections and allows the usage of the **from**-clause.

**Check any operation**

A **check** operation with no argument list allows checking whether something waits for processing in an incoming port queue. The **check** any operation allows to distinguish between different senders (in case of one-to-many connections) by using a **from** clause and to retrieve the sender by using a shorthand assignment part with a **sender** clause.

**Check on any port**

To **check** on any port, use the **any port** keywords.

**Check on any port from a port array**

To **check** on any port from a specific port array, indices use the **any from** *PortArrayRef*syntax where PortArrayRefshallbe areference to a port array identifier**.** It is also possible to store the index of a port in a single‑dimensional port array at which the operation was successful to a variable of type integer or, in case of multi‑dimensional port arrays the index of the successful port to an integer array or record of integer variable. When checking the port array for a matching message, call, reply or exception, the port indices to be checked are iterated from lowest to highest. If the port array is multi-dimensional, then the ports are iterated over from innermost to outermost array dimension from lowest to highest index for each dimension, e.g. [0][0], [0][1], [1][0], [1][1]. The first port which matches all the criteria will cause the operation to be successful even if other ports in the array would also meet the criteria.

***Restrictions***

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

1. Using the **check** operation in a wrong manner, e.g. check for an exception at a message-based port shall cause a test case error.
2. All *AddressRef* items in the **from** clause and all ValueRef items in the **sender** clause shall be of type **address**, **component** or of the address type bound to the port type (see clause C.5) of the port instance referenced in the **check** operation. No *AddressRef* in the **from** clause shall contain the special value **null** at the time of the operation.
3. The *PortArrayRef* shall be a reference to a completely initialized port array.
4. The index redirection shall only be used when the operation is used on an any from port array construct.
5. If the index redirection is used for single-dimensional arrays, the type of the integer variable shall allow storing the highest index of the respective port array.
6. If the index redirection is used for multi-dimensional arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective port array, and the its type shall allow storing the highest index (from all dimensions) of the port array.
7. If a variable referenced in the **sender** or **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **check** operation, i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **check** operation.
8. If the **check** operation contains both **from** and **sender** clause, the variable or parameter referenced in the **sender** clause shall be type compatible with the template in the **from** clause. If the operation contains a **sender** clause but no **from** clause, the sender shall be type compatible with the variable or parameter referenced in the **sender** clause.
9. The *ObjectReference* shall be of a port type.
10. The **@nodefault** modifier is allowed only in stand-alone **check** statements.

NOTE 3: In most cases the correct usage of the check operation can be checked statically, i.e. before/during compilation.

NOTE 4: An error due to a type mismatch may happen if the types in the receive part are not compatible to the types in the assignment part or, if the from clause is missing, but the type of the sender can be determined and it is not type compatible with the type in the sender clause.

***Examples***

EXAMPLE 1: Basic check

 myPort1.**check**(**receive**(5)); // Checks for an integer message of value 5.

 myPort1.**check**(**receive**(charstring:?) -> **value** v\_myCharVar);

 // Checks for a charstring message and stores the message if the message type is charstring

 myPort2.**check**(**getcall**(MyProc:{5, v\_myVar}) **from** myPartner);

 // Checks for a call of MyProc at port myPort2 from myPartner

 myPort2.**check**(**getreply**(MyProc:{5, v\_myVar} **value** 20));

 // Checks for a reply from procedure MyProc at myPort2 where the returned value is 20 and

 // the values of the two out or inout parameters are 5 and the value of v\_myVar.

 myPort2.**check**(**catch**(MyProc, s­\_myTemplate(5, v\_myVar)));

 myPort2.**check**(**getreply**(MyProc1:{?, v\_myVar} **value** \*)-> **value** v\_myReturnValue **param**(v\_myPar1,-));

 myPort.**check**(**getcall**(MyProc:{5, v\_myVar}) **from** myPartner -> **param** (v\_myPar1Var, v\_myPar2Var));

 myPort.**check**(**getcall**(MyProc:{5, v\_myVar}) -> **sender** v\_mySenderVar);

EXAMPLE 2: Check any operation

 myPort.**check**;

 myPort.**check**(**from** myPartner);

 myPort.**check**(-> **sender** v\_mySenderVar);

EXAMPLE 3: Check on any port

 **any port**.**check;**

EXAMPLE 4: Check on any port from port array

 **type** **port** MyPort **procedure** { **inout** MyProc }

 **type** **component** MyComponent {

 **port** MyPort p[10][10];

 }

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

 **any** **from** p.**check**(**catch**(MyProc, MyType:?)) -> **@index** **value** v\_i;

 // Checking for an incoming exception of the type MyType on any port of the port array p and

 // storing the index of the port on which the matching was successful first

## 23.6 The Timeout operation

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

***Syntactical Structure***

[ @nodefault ] ( *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.
7. The **@nodefault** modifier is allowed only in stand-alone **timeout** statements.

***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

### A.1.5.0 General

TTCN‑3 terminal symbols and reserved words are listed in tables A.2 and A.3.

Table A.: List of TTCN‑3 special terminal symbols

|  |  |
| --- | --- |
| Begin/end block symbols | **{ }** |
| Begin/end list symbols | **( )**  |
| Element specifier symbols | **[ ]** |
| Range symbol | **..** |
| Line and block comments | **/\* \*/**  **//** |
| Statement separator symbol | **;** |
| Arithmetic operator symbols | **+ / - \*** |
| Concatenation operator symbol | **&** |
| Relational operator symbols | **!= == >= <= < >** |
| Shift operator symbols | **<< >>** |
| Rotate operator symbols | **<@ @>** |
| String enclosure symbols | **" '**  |
| Wildcard/matching symbols | **? \***  |
| Assignment symbol | **:=**  |
| Communication operation assignment  | **->** |
| Bitstring, hexstring and Octetstring values | **B H O**  |
| Float exponent | **E** |
| List element separator symbol | **,** |
| Field reference | **.** |
| Decoded field reference | **=>** |

The predefined function identifiers defined in table 15 and described in annex C shall also be treated as reserved words.

Table A.: List of TTCN‑3 terminals which are reserved words

|  |  |  |  |
| --- | --- | --- | --- |
| **action****activate****address****alive****all****alt****altstep****and****and4b****any****anytype****bitstring****boolean****break****case****call****catch****char****charstring****check****clear****complement****component****connect****const** **continue****control****create****deactivate****decmatch****default****disconnect****display****do****done****else****encode****enumerated****error****except****exception****execute****extends****extension****external** | **fail****false****float****for****friend****from****function****getverdict****getcall****getreply****goto****group****halt****hexstring****if****ifpresent****import****in****inconc****infinity****inout****integer****interleave****kill****killed****label****language****length****log****map****match****message****mixed****mod****modifies****module****modulepar****mtc** | **noblock****none****not****not\_a\_number****not4b****nowait****null****octetstring****of****omit****on****optional****or****or4b****out****override****param****pass****pattern****permutation****port****present****private****procedure****public****raise****read****receive****record****recursive****rem****repeat****reply****return****running****runs** | **select****self****send****sender****set****setencode****setverdict****signature****start****stop****subset****superset****system****template****testcase****timeout****timer****to****trigger****true****type****union****universal****unmap****value****valueof****var****variant****verdicttype****while****with****xor****xor4b** |

The TTCN‑3 terminals listed in table A.3 shall not be used as identifiers in a TTCN‑3 module. These terminals shall be written in all lowercase letters.

Additionally, there are special TTCN-3 terminals consisting of an @-symbol, directly followed by an identifier. These terminals shall also be written in all lowercase letters.

NOTE: These terminals can be used in combination with the @-symbol, which results in a specific semantics for the annotated language element. They can also be used like any other identifier without any special meaning.

Table A.: List of TTCN‑3 terminals which are modifiers

|  |  |  |  |
| --- | --- | --- | --- |
| **@decoded****@default****@deterministic**  | **@fuzzy****@index** | **@lazy****@local** | **@nocase****@nodefault** |

Table A.: List of TTCN‑3 terminals which are reserved words in extension packages

|  |  |  |  |
| --- | --- | --- | --- |
| **apply****assert****at****configuration****conjunct****cont****delta****disjunct****duration****finished** | **history****implies****inv****mode****notinv****now****onentry****onexit** | **par****prev****realtime****seq****setstate****static****stepsize****stream** | **timestamp****until****values****wait** |

The TTCN‑3 terminals listed in table A.5 are used as keywords inside the TTCN-3 extension packages. Using these terminals in the code is not recommended as it might lead to issues in the future.

These terminals shall be written in all lowercase letters.

## A.1.6 TTCN-3 syntax BNF productions

### A.1.6.0 TTCN-3 module

TTCN3Module ::= [TTCN3ModuleKeyword](#TTTCN3ModuleKeyword) [ModuleId](#TModuleId) "{" [[ModuleDefinitionsList](#TModuleDefinitionsList)]

 "}" [[WithStatement](#TWithStatement)] [[SemiColon](#TSemiColon)]

TTCN3ModuleKeyword ::= "module"

ModuleId ::= [Identifier](#TIdentifier) [[LanguageSpec](#TLanguageSpec)]

LanguageSpec ::= [LanguageKeyword](#TLanguageKeyword) [FreeText](#TFreeText) {"," [FreeText](#TFreeText)}

LanguageKeyword ::= "language"

### A.1.6.1 Module definitions part

#### A.1.6.1.0 General

ModuleDefinitionsList ::= {[ModuleDefinition](#TModuleDefinition) [[SemiColon](#TSemiColon)]}+

ModuleDefinition ::= (([[Visibility](#TVisibility)] ([TypeDef](#TTypeDef) |

 [ConstDef](#TConstDef) |

 [TemplateDef](#TTemplateDef) |

 [ModuleParDef](#TModuleParDef) |

 [FunctionDef](#TFunctionDef) |

 [SignatureDef](#TSignatureDef) |

 [TestcaseDef](#TTestcaseDef) |

 [AltstepDef](#TAltstepDef) |

 [ImportDef](#TImportDef) |

 [ExtFunctionDef](#TExtFunctionDef) |

 [ModuleControlDef](#TModuleControlDef)

 )) |

 (["public"] [GroupDef](#TGroupDef)) |

 (["private"] [FriendModuleDef](#TFriendModuleDef))

 ) [[WithStatement](#TWithStatement)]

Visibility ::= "public" |

 "friend" |

 "private"

#### A.1.6.1.1 Typedef definitions

TypeDef ::= [TypeDefKeyword](#TTypeDefKeyword) [TypeDefBody](#TTypeDefBody)

TypeDefBody ::= [StructuredTypeDef](#TStructuredTypeDef) | [SubTypeDef](#TSubTypeDef)

TypeDefKeyword ::= "type"

StructuredTypeDef ::= [RecordDef](#TRecordDef) |

 [UnionDef](#TUnionDef) |

 [SetDef](#TSetDef) |

 [RecordOfDef](#TRecordOfDef) |

 [SetOfDef](#TSetOfDef) |

 [EnumDef](#TEnumDef) |

 [PortDef](#TPortDef) |

 [ComponentDef](#TComponentDef)

RecordDef ::= [RecordKeyword](#TRecordKeyword) [StructDefBody](#TStructDefBody)

RecordKeyword ::= "record"

StructDefBody ::= [IdentifierOrAddr](#TIdentifierOrAddr) "{" [[StructFieldDef](#TStructFieldDef)

 {"," [StructFieldDef](#TStructFieldDef)}]

 "}"

StructFieldDef ::= ([Type](#TType) | [NestedTypeDef](#TNestedTypeDef)) [Identifier](#TIdentifier) [[ArrayDef](#TArrayDef)] [[SubTypeSpec](#TSubTypeSpec)]

 [[OptionalKeyword](#TOptionalKeyword)]

NestedTypeDef ::= [NestedRecordDef](#TNestedRecordDef) |

 [NestedUnionDef](#TNestedUnionDef) |

 [NestedSetDef](#TNestedSetDef) |

 [NestedRecordOfDef](#TNestedRecordOfDef) |

 [NestedSetOfDef](#TNestedSetOfDef) |

 [NestedEnumDef](#TNestedEnumDef)

NestedRecordDef ::= [RecordKeyword](#TRecordKeyword) "{" [[StructFieldDef](#TStructFieldDef) {"," [StructFieldDef](#TStructFieldDef)}]

 "}"

NestedUnionDef ::= [UnionKeyword](#TUnionKeyword) "{" [UnionFieldDef](#TUnionFieldDef) {"," [UnionFieldDef](#TUnionFieldDef)}

 "}"

NestedSetDef ::= [SetKeyword](#TSetKeyword) "{" [[StructFieldDef](#TStructFieldDef) {"," [StructFieldDef](#TStructFieldDef)}]

 "}"

NestedRecordOfDef ::= [RecordKeyword](#TRecordKeyword) [[StringLength](#TStringLength)] [OfKeyword](#TOfKeyword) ([Type](#TType) |

 [NestedTypeDef](#TNestedTypeDef))

NestedSetOfDef ::= [SetKeyword](#TSetKeyword) [[StringLength](#TStringLength)] [OfKeyword](#TOfKeyword) ([Type](#TType) | [NestedTypeDef](#TNestedTypeDef))

NestedEnumDef ::= [EnumKeyword](#TEnumKeyword) "{" [EnumerationList](#TEnumerationList) "}"

OptionalKeyword ::= "optional"

UnionDef ::= [UnionKeyword](#TUnionKeyword) [UnionDefBody](#TUnionDefBody)

UnionKeyword ::= "union"

UnionDefBody ::= [IdentifierOrAddr](#TIdentifierOrAddr) "{" [UnionFieldDef](#TUnionFieldDef) {","

 [UnionFieldDef](#TUnionFieldDef)}

 "}"

UnionFieldDef ::= [[DefaultModifier](#TDefaultModifier)] ([Type](#TType) | [NestedTypeDef](#TNestedTypeDef)) [Identifier](#TIdentifier) [[ArrayDef](#TArrayDef)] [[SubTypeSpec](#TSubTypeSpec)]

/\*\* STATIC SEMANTICS: at most one UnionFieldDef of UnionDefBody or NestedUnionDef shall contain a DefaultModifier \*/

SetDef ::= [SetKeyword](#TSetKeyword) [StructDefBody](#TStructDefBody)

SetKeyword ::= "set"

RecordOfDef ::= [RecordKeyword](#TRecordKeyword) [[StringLength](#TStringLength)] [OfKeyword](#TOfKeyword) [StructOfDefBody](#TStructOfDefBody)

OfKeyword ::= "of"

StructOfDefBody ::= ([Type](#TType) | [NestedTypeDef](#TNestedTypeDef)) [IdentifierOrAddr](#TIdentifierOrAddr)

 [[SubTypeSpec](#TSubTypeSpec)]

SetOfDef ::= [SetKeyword](#TSetKeyword) [[StringLength](#TStringLength)] [OfKeyword](#TOfKeyword) [StructOfDefBody](#TStructOfDefBody)

EnumDef ::= [EnumKeyword](#TEnumKeyword) [IdentifierOrAddr](#TIdentifierOrAddr) "{" [EnumerationList](#TEnumerationList)

 "}"

EnumKeyword ::= "enumerated"

EnumerationList ::= [Enumeration](#TEnumeration) {"," [Enumeration](#TEnumeration)}

Enumeration ::= [Identifier](#TIdentifier) ["(" [IntegerValueOrRange](#TEnumValueOrRange) {"," [IntegerValueOrRange](#TEnumValueOrRange) } ")"]

IntegerValueOrRange ::= [IntegerValue](#TIntegerValue) [".." [IntegerValue](#TIntegerValue)]

IntegerValue ::= [[Minus](#TMinus)] [Number](#TNumber)

SubTypeDef ::= [Type](#TType) [IdentifierOrAddr](#TIdentifierOrAddr) [[ArrayDef](#TArrayDef)] [[SubTypeSpec](#TSubTypeSpec)]

SubTypeSpec ::= [AllowedValuesSpec](#TAllowedValuesSpec) [[StringLength](#TStringLength)] | [StringLength](#TStringLength)

/\* STATIC SEMANTICS - AllowedValues shall be of the same type as the field being subtyped \*/

AllowedValuesSpec ::= "(" (([TemplateOrRange](#TTemplateOrRange) {"," [TemplateOrRange](#TTemplateOrRange)}) |

 [CharStringMatch](#TCharStringMatch)) ")"

TemplateOrRange ::= [RangeDef](#TRangeDef) |

 [TemplateBody](#TTemplateBody) |

 [Type](#TType)

/\* STATIC SEMANTICS - RangeDef production shall only be used with integer, charstring, universal charstring or float based types \*/

/\* STATIC SEMANTICS - When subtyping charstring or universal charstring range and values shall not be mixed in the same SubTypeSpec \*/

RangeDef ::= [Bound](#TBound) ".." [Bound](#TBound)

StringLength ::= [LengthKeyword](#TLengthKeyword) "(" [SingleExpression](#TSingleExpression) [".."(SingleExpression | InfinityKeyword) ] ")"

/\* STATIC SEMANTICS - StringLength shall only be used with String types or to limit set of and record of. SingleExpression and Bound shall evaluate to non-negative integer values (in case of Bound including infinity) \*/

LengthKeyword ::= "length"

PortDef ::= [PortKeyword](#TPortKeyword) [PortDefBody](#TPortDefBody)

PortDefBody ::= [Identifier](#TIdentifier) [PortDefAttribs](#TPortDefAttribs)

PortKeyword ::= "port"

PortDefAttribs ::= [MessageAttribs](#TMessageAttribs) |

 [ProcedureAttribs](#TProcedureAttribs) |

 [MixedAttribs](#TMixedAttribs)

MessageAttribs ::= [MessageKeyword](#TMessageKeyword) "{" {([AddressDecl](#TAddressDecl) |

 [MessageList](#TMessageList) |

 [ConfigParamDef](#TConfigParamDef)

 ) [[SemiColon](#TSemiColon)]}+ "}"

ConfigParamDef ::= [MapParamDef](#TMapParamDef) | [UnmapParamDef](#TUnmapParamDef)

MapParamDef ::= [MapKeyword](#TMapKeyword) [ParamKeyword](#TParamKeyword) "(" [FormalValuePar](#TFormalValuePar) {"," [FormalValuePar](#TFormalValuePar)}

 ")"

UnmapParamDef ::= [UnmapKeyword](#TUnmapKeyword) [ParamKeyword](#TParamKeyword) "(" [FormalValuePar](#TFormalValuePar) {","

 [FormalValuePar](#TFormalValuePar)}

 ")"

AddressDecl ::= [AddressKeyword](#TAddressKeyword) [Type](#TType)

MessageList ::= [Direction](#TDirection) [AllOrTypeList](#TAllOrTypeList)

Direction ::= [InParKeyword](#TInParKeyword) |

 [OutParKeyword](#TOutParKeyword) |

 [InOutParKeyword](#TInOutParKeyword)

MessageKeyword ::= "message"

AllOrTypeList ::= [AllKeyword](#TAllKeyword) | [TypeList](#TTypeList)

/\* NOTE: The use of AllKeyword in port definitions is deprecated \*/

AllKeyword ::= "all"

TypeList ::= [Type](#TType) {"," [Type](#TType)}

ProcedureAttribs ::= [ProcedureKeyword](#TProcedureKeyword) "{" {([AddressDecl](#TAddressDecl) |

 [ProcedureList](#TProcedureList) |

 [ConfigParamDef](#TConfigParamDef)

 ) [[SemiColon](#TSemiColon)]}+ "}"

ProcedureKeyword ::= "procedure"

ProcedureList ::= [Direction](#TDirection) [AllOrSignatureList](#TAllOrSignatureList)

AllOrSignatureList ::= [AllKeyword](#TAllKeyword) | [SignatureList](#TSignatureList)

SignatureList ::= [Signature](#TSignature) {"," [Signature](#TSignature)}

MixedAttribs ::= [MixedKeyword](#TMixedKeyword) "{" {([AddressDecl](#TAddressDecl) |

 [MixedList](#TMixedList) |

 [ConfigParamDef](#TConfigParamDef)

 ) [[SemiColon](#TSemiColon)]}+ "}"

MixedKeyword ::= "mixed"

MixedList ::= [Direction](#TDirection) [ProcOrTypeList](#TProcOrTypeList)

ProcOrTypeList ::= [AllKeyword](#TAllKeyword) | ([ProcOrType](#TProcOrType) {"," [ProcOrType](#TProcOrType)})

ProcOrType ::= [Signature](#TSignature) | [Type](#TType)

ComponentDef ::= [ComponentKeyword](#TComponentKeyword) [Identifier](#TIdentifier) [[ExtendsKeyword](#TExtendsKeyword) [ComponentType](#TComponentType)

 {"," [ComponentType](#TComponentType)}] "{"

 [[ComponentDefList](#TComponentDefList)] "}"

ComponentKeyword ::= "component"

ExtendsKeyword ::= "extends"

ComponentType ::= [ExtendedIdentifier](#TExtendedIdentifier)

ComponentDefList ::= {[ComponentElementDef](#TComponentElementDef) [[WithStatement](#TWithStatement)] [[SemiColon](#TSemiColon)]}

ComponentElementDef ::= [PortInstance](#TPortInstance) |

 [VarInstance](#TVarInstance) |

 [TimerInstance](#TTimerInstance) |

 [ConstDef](#TConstDef) |

 [TemplateDef](#TTemplateDef)

PortInstance ::= [PortKeyword](#TPortKeyword) [ExtendedIdentifier](#TExtendedIdentifier) [PortElement](#TPortElement) {"," [PortElement](#TPortElement)}

PortElement ::= [Identifier](#TIdentifier) [[ArrayDef](#TArrayDef)]

#### A.1.6.1.2 Constant definitions

ConstDef ::= [ConstKeyword](#TConstKeyword) [Type](#TType) [ConstList](#TConstList)

ConstList ::= [SingleConstDef](#TSingleConstDef) {"," [SingleConstDef](#TSingleConstDef)}

SingleConstDef ::= [Identifier](#TIdentifier) [[ArrayDef](#TArrayDef)] [AssignmentChar](#TAssignmentChar) [ConstantExpression](#TConstantExpression)

ConstKeyword ::= "const"

#### A.1.6.1.3 Template definitions

TemplateDef ::= [TemplateKeyword](#TTemplateKeyword) [[TemplateRestriction](#TTemplateRestriction)] [[FuzzyModifier](#TFuzzyModifier) [DeterministicModifier]]

 [BaseTemplate](#TBaseTemplate) [[DerivedDef](#TDerivedDef)] [AssignmentChar](#TAssignmentChar) Base[TemplateBody](#TTemplateBody)

BaseTemplate ::= ([Type](#TType) | [Signature](#TSignature)) [Identifier](#TIdentifier) ["(" [TemplateOrValueFormalParList](#TTemplateOrValueFormalParList)

 ")"]

TemplateKeyword ::= "template"

DerivedDef ::= [ModifiesKeyword](#TModifiesKeyword) [ExtendedIdentifier](#TExtendedIdentifier)

ModifiesKeyword ::= "modifies"

TemplateOrValueFormalParList ::= [TemplateOrValueFormalPar](#TTemplateOrValueFormalPar) {"," [TemplateOrValueFormalPar](#TTemplateOrValueFormalPar)}

TemplateOrValueFormalPar ::= [FormalValuePar](#TFormalValuePar) | [FormalTemplatePar](#TFormalTemplatePar)

/\* STATIC SEMANTICS - FormalValuePar shall resolve to an in parameter \*/

TemplateBody ::= [DerivedTemplateBody](#TDerivedTemplateBody) | [BaseTemplateBody](#TBaseTemplateBody)

BaseTemplateBody ::= ([SimpleSpec](#TSimpleSpec) |

 [FieldSpecList](#TFieldSpecList) |

 [ArrayValueOrAttrib](#TArrayValueOrAttrib)

 ) [[ExtraMatchingAttributes](#TExtraMatchingAttributes)]

/\* STATIC SEMANTICS - Within BaseTeplateBody the ArrayValueOrAttrib can be used for array, record, record of and set of types. \*/

SimpleSpec ::= ([SingleExpression](#TSingleExpression) ["&" [SimpleTemplateSpec](#TSimpleTemplateSpec)]) | [SimpleTemplateSpec](#TSimpleTemplateSpec)

SimpleTemplateSpec ::= [SingleTemplateExpression](#TSingleTemplateExpression) ["&" [SimpleSpec](#TSimpleSpec)]

SingleTemplateExpression ::= [MatchingSymbol](#TMatchingSymbol) |

 ({[TemplateRefWithParList](#TTemplateRefWithParList) [[ExtendedFieldReference](#TExtendedFieldReference)]) |

 [ExtendedIdentifier](#TExtendedIdentifier) [EnumTemplateExtension](#TEnumTemplateExtension)

/\*\* STATIC Semantics: ExtendedIdentifier shall refer to an enumerated value with associated value \*/

EnumTemplateExtension ::= "(" ([BaseTemplateBody](#TTemplateBody) | [Range](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C05PWRTCH%5CCR7709_v3.docx#TRange)) {"," ([BaseTemplateBody](#TTemplateBody) | [Range](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C05PWRTCH%5CCR7709_v3.docx#TRange)) } ")"

/\*\* STATIC Semantics: each TemplateBody shall be an integer template template and the limits of each Range an integer value\*/

FieldSpecList ::= "{" [FieldSpec](#TFieldSpec) {"," [FieldSpec](#TFieldSpec)} "}"

FieldSpec ::= [FieldReference](#TFieldReference) [AssignmentChar](#TAssignmentChar) ([TemplateBody](#TTemplateBody) | [Minus](#TMinus))

FieldReference ::= [StructFieldRef](#TStructFieldRef) |

 [ArrayOrBitRef](#TArrayOrBitRef) |

 [ParRef](#TParRef)

StructFieldRef ::= [Identifier](#TIdentifier) |

 [PredefinedType](#TPredefinedType) |

 [TypeReference](#TTypeReference)

/\* STATIC SEMANTICS - PredefinedType and TypeReference shall be used for anytype value notation only. PredefinedType shall not be AnyTypeKeyword.\*/

ParRef ::= [Identifier](#TIdentifier)

/\* STATIC SEMANTICS - Identifier in ParRef shall be a formal parameter identifier from the associated signature definition \*/

ArrayOrBitRef ::= "[" [FieldOrBitNumber](#TFieldOrBitNumber) "]"

/\* STATIC SEMANTICS - ArrayRef shall be optionally used for array types and TTCN-3 record of and set of. The same notation can be used for a Bit reference inside an TTCN-3 charstring, universal charstring, bitstring, octetstring and hexstring type \*/

FieldOrBitNumber ::= [SingleExpression](#TSingleExpression)

/\* STATIC SEMANTICS - SingleExpression will resolve to a value of integer type \*/

ArrayValueOrAttrib ::= "{" [[ArrayElementSpecList](#TArrayElementSpecList)] "}"

ArrayElementSpecList ::= [ArrayElementSpec](#TArrayElementSpec) {"," [ArrayElementSpec](#TArrayElementSpec)}

ArrayElementSpec ::= [Minus](#TMinus) |

 [PermutationMatch](#TPermutationMatch) |

 [TemplateBody](#TTemplateBody)

MatchingSymbol ::= [Complement](#TComplement) |

 ([AnyValue](#TAnyValue) [[WildcardLengthMatch](#TWildcardLengthMatch)]) |

 ([AnyOrOmit](#TAnyOrOmit) [[WildcardLengthMatch](#TWildcardLengthMatch)]) |

 [ListOfTemplates](#TListOfTemplates) |

 [Range](#TRange) |

 [BitStringMatch](#TBitStringMatch) |

 [HexStringMatch](#THexStringMatch) |

 [OctetStringMatch](#TOctetStringMatch) |

 [CharStringMatch](#TCharStringMatch) |

 [SubsetMatch](#TSubsetMatch) |

 [SupersetMatch](#TSupersetMatch) |

 [DecodedContentMatch](#TDecodedContentMatch)

DecodedContentMatch ::= [DecodedMatchKeyword](#TDecodedMatchKeyword) ["(" [[Expression](#TExpression)] ")"] [TemplateInstance](#TTemplateInstance)

DecodedMatchKeyword ::= "decmatch"

/\* STATIC SEMANTIC – WildcardLengthMatch shall be used when MatchingSymbol is used in fractions of a concatenated string or list (see clause 15.11) and shall not be used in other cases. In this case, the Complement, ListOfTemplates, Range, BitStringMatch, HexStringMatch, OctetStringMatch, CharStringMatch, SubsetMatch and SupersetMatch productions shall not be used. \*/

ExtraMatchingAttributes ::= [StringLength](#TStringLength) |

 [IfPresentKeyword](#TIfPresentKeyword) |

 ([StringLength](#TStringLength) [IfPresentKeyword](#TIfPresentKeyword))

BitStringMatch ::= "'" {[BinOrMatch](#TBinOrMatch)} "'" "B"

BinOrMatch ::= [Bin](#TBin) |

 [AnyValue](#TAnyValue) |

 [AnyOrOmit](#TAnyOrOmit)

HexStringMatch ::= "'" {[HexOrMatch](#THexOrMatch)} "'" "H"

HexOrMatch ::= [Hex](#THex) |

 [AnyValue](#TAnyValue) |

 [AnyOrOmit](#TAnyOrOmit)

OctetStringMatch ::= "'" {[OctOrMatch](#TOctOrMatch)} "'" "O"

OctOrMatch ::= [Oct](#TOct) |

 [AnyValue](#TAnyValue) |

 [AnyOrOmit](#TAnyOrOmit)

CharStringMatch ::= [PatternKeyword](#TPatternKeyword) [[CaseInsenModifier](#TCaseInsenModifier)] [PatternParticle](#TPatternParticle) {"&" [PatternParticle](#TPatternParticle)}

PatternParticle ::= [Pattern](#TPattern) | [ReferencedValue](#TReferencedValue)

PatternKeyword ::= "pattern"

Pattern ::= """ {[PatternElement](#TPatternElement)} """

PatternElement ::= (("\" ("?" | "\*" | "\" | "[" | "]" | "{" | "}" |

 """ | "|" | "(" | ")" | "#" | "+" | "d" |

 "w" | "t" | "n" | "r" | "s" | "b"

 )) | ("?" | "\*" | "\" | "|" | "+"

 ) | ("[" ["^"] [{[PatternClassChar](#TPatternClassChar) ["-"

 [PatternClassChar](#TPatternClassChar)]}]

 "]") |

 ("{" ["\"] [ReferencedValue](#TReferencedValue) "}") | ("\" "N" "{"

 ([ReferencedValue](#TReferencedValue) |

 [Type](#TType)) "}") |

 (""" """) |

 ("(" [PatternElement](#TPatternElement) ")") |

 ("#" ([Num](#TNum) |

 ("(" [Number](#TNumber) "," [[Number](#TNumber)] ")") |

 ("(" "," [Number](#TNumber) ")") |

 ("(" [","] ")") [Num](#TNum) ")"

 ))

 ) | [PatternChar](#TPatternChar)

PatternChar ::= [NonSpecialPatternChar](#TNonSpecialPatternChar) | [PatternQuadruple](#TPatternQuadruple)

/\* STATIC SEMANTICS: Characters "?", "\*", "\", "[", "]", "{", "}", """, "|", "(", ")", "#", "+", "d", "^", "N" have special semantics – they are metacharacters for the definition of pattern elements – only if they follow the BNF as defined above, if not they are interpreted like normal characters \*/

NonSpecialPatternChar ::= [Char](#TChar)

PatternClassChar ::= [NonSpecialPatternClassChar](#TNonSpecialPatternClassChar) |

 [PatternQuadruple](#TPatternQuadruple) |

 "\" [EscapedPatternClassChar](#TEscapedPatternClassChar)

NonSpecialPatternClassChar ::= [Char](#TChar)

/\* STATIC SEMANTICS: Characters "[", "-", "^", "]", "\", "q", ","have special semantics – they are metacharacters for the definition of pattern class characters – only if they follow the BNF as defined above, if not they are interpreted like normal characters \*/

EscapedPatternClassChar ::= "[" | "-" | "^" | "]"

PatternQuadruple ::= "\" "q" "(" [Number](#TNumber) "," [Number](#TNumber) "," [Number](#TNumber) ","

 [Number](#TNumber) ")"

Complement ::= [ComplementKeyword](#TComplementKeyword) [ListOfTemplates](#TListOfTemplates)

ComplementKeyword ::= "complement"

ListOfTemplates ::= "(" [TemplateListItem](#TTemplateListItem) {"," [TemplateListItem](#TTemplateListItem)} ")"

TemplateListItem ::= [TemplateBody](#TTemplateBody) | [AllElementsFrom](#TAllElementsFrom)

AllElementsFrom ::= [AllKeyword](#TAllKeyword) [FromKeyword](#TFromKeyword) [TemplateBody](#TTemplateBody)

SubsetMatch ::= [SubsetKeyword](#TSubsetKeyword) [ListOfTemplates](#TListOfTemplates)

SubsetKeyword ::= "subset"

SupersetMatch ::= [SupersetKeyword](#TSupersetKeyword) [ListOfTemplates](#TListOfTemplates)

SupersetKeyword ::= "superset"

PermutationMatch ::= [PermutationKeyword](#TPermutationKeyword) [ListOfTemplates](#TListOfTemplates)

/\* STATIC SEMANTICS: Restrictions on the content of TemplateBody within the ListOfTemplates are given in clause B.1.3.3. \*/

PermutationKeyword ::= "permutation"

AnyValue ::= "?"

AnyOrOmit ::= "\*"

WildcardLengthMatch ::= [LengthKeyword](#TLengthKeyword) "(" [SingleExpression](#TSingleExpression) ")"

/\* STATIC SEMANTICS: SingleExpression shall evaluate to type integer \*/

IfPresentKeyword ::= "ifpresent"

PresentKeyword ::= "present"

Range ::= "(" [Bound](#TBound) ".." [Bound](#TBound) ")"

Bound ::= (["!"] [SingleExpression](#TSingleExpression)) | ([[Minus](#TMinus)] [InfinityKeyword](#TInfinityKeyword))

/\* STATIC SEMANTICS - Bounds shall evaluate to types integer, charstring, universal charstring or float. In case they evaluate to types charstring or universal charstring, the string length shall be 1. infinity as lower bound and –infinity as upper bound are allowed for float types only. \*/

InfinityKeyword ::= "infinity"

ActualParAssignment ::= [Identifier](#TIdentifier) ":=" [TemplateInstance](#TInLineTemplate)

/\* STATIC SEMANTICS – if a value parameter is used, an in-line template shall evaluate to a value \*/ TemplateRefWithParList ::= [ExtendedIdentifier](#TExtendedIdentifier) [[ActualParList](#TActualParList)]

TemplateInstance ::= [([Type](#TType) | [Signature](#TSignature)) [Colon](#TColon)]

[TemplateBody](#TTemplateBody)

DerivedTemplateBody ::= [ModifiesKeyword](#TModifiesKeyword) [BaseTemplateBody](#TBaseTemplateBody) [AssignmentChar](#TAssignmentChar) [BaseTemplateBody](#TBaseTemplateBody)

ActualParList ::= "(" [([ActualPar](#TActualPar) {"," [ActualPar](#TActualPar) })

 {"," [ActualParAssignment](#TActualParAssignment)} |

 ([ActualParAssignment](#TActualParAssignment) {"," [ActualParAssignment](#TActualParAssignment)})]

 ")"

ActualPar ::= [TemplateInstance](#TInLineTemplate) | [Minus](#TMinus)

/\* STATIC SEMANTICS - When the corresponding formal parameter is not of template type the TemplateInstance production shall resolve to one or more SingleExpressions \*/

TemplateOps ::= [MatchOp](#TMatchOp) | [ValueofOp](#TValueofOp)

MatchOp ::= [MatchKeyword](#TMatchKeyword) "(" [Expression](#TExpression) "," [TemplateInstance](#TInLineTemplate) ")"

MatchKeyword ::= "match"

ValueofOp ::= [ValueofKeyword](#TValueofKeyword) "(" [TemplateInstance](#TTemplateInstance)")"

ValueofKeyword ::= "valueof"

#### A.1.6.1.4 Function definitions

FunctionDef ::= [FunctionKeyword](#TFunctionKeyword) [ [DeterministicModifier](#TDeterministicModifier) | [ControlModifier](#TControlModifier) ]

 [IdentifierOrControl](#TIdentifierOrControl)

 "(" [[FunctionFormalParList](#TFunctionFormalParList)] ")" [[RunsOnSpec](#TRunsOnSpec)] [[MtcSpec](#TMtcSpec)]

 [[SystemSpec](#TSystemSpec)] [[ReturnType](#TReturnType)] [StatementBlock](#TStatementBlock)

FunctionKeyword ::= "function"

FunctionFormalParList ::= [FunctionFormalPar](#TFunctionFormalPar) {"," [FunctionFormalPar](#TFunctionFormalPar)}

FunctionFormalPar ::= [FormalValuePar](#TFormalValuePar) |

 [FormalTemplatePar](#TFormalTemplatePar)

ReturnType ::= [ReturnKeyword](#TReturnKeyword) [[TemplateKeyword](#TTemplateKeyword) | [RestrictedTemplate](#TRestrictedTemplate)]

 [Type](#TType) [[ArrayDef](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C0EG46CRK%5CCR7496-v1.docx#TArrayDef)]

ReturnKeyword ::= "return"

RunsOnSpec ::= [RunsKeyword](#TRunsKeyword) [OnKeyword](#TOnKeyword) [ComponentType](#TComponentType)

RunsKeyword ::= "runs"

OnKeyword ::= "on"

MtcSpec ::= [MTCKeyword](#TMTCKeyword) [ComponentType](#TComponentType)

MTCKeyword ::= "mtc"

StatementBlock ::= "{" [[FunctionDefOrStatementList](#TFunctionStatementList)] "}"

FunctionDefOrStatementList ::= {( [FunctionDef](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5CC3T541B1%5CCR7624-2.docx#TFunctionDefList) | [FunctionStatement](#TFunctionStatement)) [[SemiColon](#TSemiColon)]}+

FunctionDef ::= ([FunctionLocalDef](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5CC3T541B1%5CCR7624-2.docx#TFunctionLocalDef) | [FunctionLocalInst](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5CC3T541B1%5CCR7624-2.docx#TFunctionLocalInst)) [[WithStatement](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5CC3T541B1%5CCR7624-2.docx#TWithStatement)]

FunctionLocalInst ::= [VarInstance](#TVarInstance) | [TimerInstance](#TTimerInstance)

FunctionLocalDef ::= [ConstDef](#TConstDef) | [TemplateDef](#TTemplateDef)

FunctionStatement ::= [ConfigurationStatements](#TConfigurationStatements) |

 [TimerStatements](#TTimerStatements) |

 [CommunicationStatements](#TCommunicationStatements) |

 [BasicStatements](#TBasicStatements) |

 [BehaviourStatements](#TBehaviourStatements) |

 [SetLocalVerdict](#TSetLocalVerdict) |

 [SUTStatements](#TSUTStatements) |

 [TestcaseOperation](#TTestcaseOperation)

FunctionInstance ::= [FunctionRef](#TFunctionRef) [ "(" [[ActualParList](#TActualParList)] ")" ]

/\* STATIC SEMANTICS – the part is only optional if the FunctionRef uses the ControlKeyword and the referenced control function has no formal parameters \*/

FunctionRef ::= [[Identifier](#TIdentifier) [Dot](#TDot)] ([Identifier](#TIdentifier) | [PreDefFunctionIdentifier](#TPreDefFunctionIdentifier) | [ControlKeyword](#TControlKeyword) )

PreDefFunctionIdentifier ::= [Identifier](#TIdentifier) [[CaseInsenModifier](#TCaseInsenModifier)]

/\* STATIC SEMANTICS - The Identifier shall be one of the pre-definedpredefined TTCN-3 function identifiers from Annex C of ES 201 873-1. CaseInsenModifier shall be present only if Identifier is "regexp". \*/

/\* STATIC SEMANTICS – if a value parameter is used, an in-line template shall evaluate to a value \*/

#### A.1.6.1.5 Signature definitions

SignatureDef ::= [SignatureKeyword](#TSignatureKeyword) [Identifier](#TIdentifier) "(" [[SignatureFormalParList](#TSignatureFormalParList)]

 ")" [[ReturnType](#TReturnType) | [NoBlockKeyword](#TNoBlockKeyword)] [[ExceptionSpec](#TExceptionSpec)]

SignatureKeyword ::= "signature"

SignatureFormalParList ::= [FormalValuePar](#TFormalValuePar) {"," [FormalValuePar](#TFormalValuePar)}

ExceptionSpec ::= [ExceptionKeyword](#TExceptionKeyword) "(" [TypeList](#TTypeList) ")"

ExceptionKeyword ::= "exception"

Signature ::= [ExtendedIdentifier](#TExtendedIdentifier)

NoBlockKeyword ::= "noblock"

#### A.1.6.1.6 Testcase definitions

TestcaseDef ::= [TestcaseKeyword](#TTestcaseKeyword) [Identifier](#TIdentifier) "(" [[TemplateOrValueFormalParList](#TTemplateOrValueFormalParList)]

 ")" [ConfigSpec](#TConfigSpec) [StatementBlock](#TStatementBlock)

TestcaseKeyword ::= "testcase"

ConfigSpec ::= [RunsOnSpec](#TRunsOnSpec) [[SystemSpec](#TSystemSpec)]

SystemSpec ::= [SystemKeyword](#TSystemKeyword) [ComponentType](#TComponentType)

SystemKeyword ::= "system"

TestcaseInstance ::= [ExecuteKeyword](#TExecuteKeyword) "(" [ExtendedIdentifier](#TExtendedIdentifier) "(" [[ActualParList](#TActualParList)]

 ")" ["," ([Expression](#TExpression) | [Minus](#TMinus)) ["," [SingleExpression](#TSingleExpression)]]

 ")"

ExecuteKeyword ::= "execute"

#### A.1.6.1.7 Altstep definitions

AltstepDef ::= [AltstepKeyword](#TAltstepKeyword) [ [ControlModifier](#TControlModifier) ] [Identifier](#TIdentifier) "(" [[FunctionFormalParList](#TFunctionFormalParList)]

 ")" [[RunsOnSpec](#TRunsOnSpec)] [[MtcSpec](#TMtcSpec)] [[SystemSpec](#TSystemSpec)] "{" [AltstepLocalDefList](#TAltstepLocalDefList)

 [AltGuardList](#TAltGuardList) "}"

AltstepKeyword ::= "altstep"

AltstepLocalDefList ::= {[AltstepLocalDef](#TAltstepLocalDef) [[WithStatement](#TWithStatement)] [[SemiColon](#TSemiColon)]}

AltstepLocalDef ::= [VarInstance](#TVarInstance) |

 [TimerInstance](#TTimerInstance) |

 [ConstDef](#TConstDef) |

 [TemplateDef](#TTemplateDef)

AltstepInstance ::= [ExtendedIdentifier](#TExtendedIdentifier) "(" [[ActualParList](#TActualParList)]

 ")"

#### A.1.6.1.8 Import definitions

ImportDef ::= [ImportKeyword](#TImportKeyword) [ImportFromSpec](#TImportFromSpec) [PortRedirectSymbol Identifier]

 ([AllWithExcepts](#TAllWithExcepts) | ("{" [ImportSpec](#TImportSpec) "}"))

ImportKeyword ::= "import"

AllWithExcepts ::= [AllKeyword](#TAllKeyword) [[ExceptsDef](#TExceptsDef)]

ExceptsDef ::= [ExceptKeyword](#TExceptKeyword) "{" [ExceptSpec](#TExceptSpec) "}"

ExceptKeyword ::= "except"

ExceptSpec ::= {[ExceptElement](#TExceptElement) [[SemiColon](#TSemiColon)]}

ExceptElement ::= [ExceptGroupSpec](#TExceptGroupSpec) |

 [ExceptTypeDefSpec](#TExceptTypeDefSpec) |

 [ExceptTemplateSpec](#TExceptTemplateSpec) |

 [ExceptConstSpec](#TExceptConstSpec) |

 [ExceptTestcaseSpec](#TExceptTestcaseSpec) |

 [ExceptAltstepSpec](#TExceptAltstepSpec) |

 [ExceptFunctionSpec](#TExceptFunctionSpec) |

 [ExceptSignatureSpec](#TExceptSignatureSpec) |

 [ExceptModuleParSpec](#TExceptModuleParSpec)

ExceptGroupSpec ::= [GroupKeyword](#TGroupKeyword) ([QualifiedIdentifierList](#TQualifiedIdentifierList) | [AllKeyword](#TAllKeyword))

IdentifierListOrAll ::= [IdentifierList](#TIdentifierList) | [AllKeyword](#TAllKeyword)

TypeIdListOrAll ::= [TypeIdentifierList](#TTypeIdentifierList) | [AllKeyword](#TAllKeyword)

FuncIdListOrAll ::= [FuncIdentifierList](#TFuncIdentifierList) | [AllKeyword](#TAllKeyword)

ExceptTypeDefSpec ::= [TypeDefKeyword](#TTypeDefKeyword) [TypeIdListOrAll](#TTypeIdListOrAll)

ExceptTemplateSpec ::= [TemplateKeyword](#TTemplateKeyword) [IdentifierListOrAll](#TIdentifierListOrAll)

ExceptConstSpec ::= [ConstKeyword](#TConstKeyword) [IdentifierListOrAll](#TIdentifierListOrAll)

ExceptTestcaseSpec ::= [TestcaseKeyword](#TTestcaseKeyword) [IdentifierListOrAll](#TIdentifierListOrAll)

ExceptAltstepSpec ::= [AltstepKeyword](#TAltstepKeyword) [IdentifierListOrAll](#TIdentifierListOrAll)

ExceptFunctionSpec ::= [FunctionKeyword](#TFunctionKeyword) [FuncIdListOrAll](#TFuncIdListOrAll)

ExceptSignatureSpec ::= [SignatureKeyword](#TSignatureKeyword) [IdentifierListOrAll](#TIdentifierListOrAll)

ExceptModuleParSpec ::= [ModuleParKeyword](#TModuleParKeyword) [IdentifierListOrAll](#TIdentifierListOrAll)

ImportSpec ::= {[ImportElement](#TImportElement) [[SemiColon](#TSemiColon)]}

ImportElement ::= [ImportGroupSpec](#TImportGroupSpec) |

 [ImportTypeDefSpec](#TImportTypeDefSpec) |

 [ImportTemplateSpec](#TImportTemplateSpec) |

 [ImportConstSpec](#TImportConstSpec) |

 [ImportTestcaseSpec](#TImportTestcaseSpec) |

 [ImportAltstepSpec](#TImportAltstepSpec) |

 [ImportFunctionSpec](#TImportFunctionSpec) |

 [ImportSignatureSpec](#TImportSignatureSpec) |

 [ImportModuleParSpec](#TImportModuleParSpec) |

 [ImportImportSpec](#TImportImportSpec)

ImportFromSpec ::= [FromKeyword](#TFromKeyword) [ModuleId](#TModuleId)

ImportGroupSpec ::= [GroupKeyword](#TGroupKeyword) ([GroupRefListWithExcept](#TGroupRefListWithExcept) | [AllGroupsWithExcept](#TAllGroupsWithExcept))

GroupRefListWithExcept ::= [QualifiedIdentifierWithExcept](#TQualifiedIdentifierWithExcept) {"," [QualifiedIdentifierWithExcept](#TQualifiedIdentifierWithExcept)}

AllGroupsWithExcept ::= [AllKeyword](#TAllKeyword) [[ExceptKeyword](#TExceptKeyword) [QualifiedIdentifierList](#TQualifiedIdentifierList)]

QualifiedIdentifierWithExcept ::= [QualifiedIdentifier](#TQualifiedIdentifier) [[ExceptsDef](#TExceptsDef)]

IdentifierListOrAllWithExcept ::= [IdentifierList](#TIdentifierList) | [AllWithExcept](#TAllWithExcept)

TypeIdListOrAllWithExcept ::= [TypeIdentifierList](#TTypeIdentifierList) | [AllTypesExcept](#TAllTypesExcept)

FuncIdListOrAllWithExcept ::= [FuncIdentifierList](#TFuncIdentifierList) | [AllFunctionsExcept](#TAllFunctionsExcept)

ImportTypeDefSpec ::= [TypeDefKeyword](#TTypeDefKeyword) [TypeIdListOrAllWithExcept](#TTypeIdListOrAllWithExcept)

AllWithExcept ::= [AllKeyword](#TAllKeyword) [[ExceptKeyword](#TExceptKeyword) [IdentifierList](#TIdentifierList)]

AllTypesExcept ::= [AllKeyword](#TAllKeyword) [[ExceptKeyword](#TExceptKeyword) [TypeIdentifierList](#TTypeIdentifierList)]

AllFunctionsExcept ::= [AllKeyword](#TAllKeyword) [[ExceptKeyword](#TExceptKeyword) [FuncIdentifierList](#TFuncIdentifierList)]

ImportTemplateSpec ::= [TemplateKeyword](#TTemplateKeyword) [IdentifierListOrAllWithExcept](#TIdentifierListOrAllWithExcept)

ImportConstSpec ::= [ConstKeyword](#TConstKeyword) [IdentifierListOrAllWithExcept](#TIdentifierListOrAllWithExcept)

ImportAltstepSpec ::= [AltstepKeyword](#TAltstepKeyword) [IdentifierListOrAllWithExcept](#TIdentifierListOrAllWithExcept)

ImportTestcaseSpec ::= [TestcaseKeyword](#TTestcaseKeyword) [IdentifierListOrAllWithExcept](#TIdentifierListOrAllWithExcept)

ImportFunctionSpec ::= [FunctionKeyword](#TFunctionKeyword) [FuncIdListOrAllWithExcept](#TFuncIdListOrAllWithExcept)

ImportSignatureSpec ::= [SignatureKeyword](#TSignatureKeyword) [IdentifierListOrAllWithExcept](#TIdentifierListOrAllWithExcept)

ImportModuleParSpec ::= [ModuleParKeyword](#TModuleParKeyword) [IdentifierListOrAllWithExcept](#TIdentifierListOrAllWithExcept)

ImportImportSpec ::= [ImportKeyword](#TImportKeyword) [AllKeyword](#TAllKeyword)

TypeIdentifierList ::= [IdentifierOrAddr](#TIdentifierOrAddr) {"," [IdentifierOrAddr](#TIdentifierOrAddr) }

IdentifierOrAddr ::= [Identifier](#TIdentifier) | [AddressKeyword](#TAddressKeyword)

FuncIdentifierList ::= [IdentifierOrControl](#TIdentifierOrControl) {"," [IdentifierOrControl](#TIdentifierOrControl) }

IdentifierOrControl ::= [Identifier](#TIdentifier) | [ControlKeyword](#TControlKeyword)

#### A.1.6.1.9 Group definitions

GroupDef ::= [GroupKeyword](#TGroupKeyword) [Identifier](#TIdentifier) "{" [[ModuleDefinitionsList](#TModuleDefinitionsList)] "}"

GroupKeyword ::= "group"

#### A.1.6.1.10 External function definitions

ExtFunctionDef ::= [ExtKeyword](#TExtKeyword) [FunctionKeyword](#TFunctionKeyword) [[DeterministicModifier](#TDeterministicModifier)]

 [Identifier](#TIdentifier) "(" [[FunctionFormalParList](#TFunctionFormalParList)] ")" [[ReturnType](#TReturnType)]

ExtKeyword ::= "external"

#### A.1.6.1.11 Void

#### A.1.6.1.12 Module parameter definitions

ModuleParDef ::= [ModuleParKeyword](#TModuleParKeyword) ([ModulePar](#TModulePar) | ("{" [MultitypedModuleParList](#TMultitypedModuleParList)

 "}"))

ModuleParKeyword ::= "modulepar"

MultitypedModuleParList ::= {[ModulePar](#TModulePar) [[SemiColon](#TSemiColon)]}

ModulePar ::= [ ([TemplateKeyword](#TTemplateKeyword) | [RestrictedTemplate](#TRestrictedTemplate)) ] [Type](#TType) [ModuleParList](#TModuleParList)

ModuleParList ::= [Identifier](#TIdentifier) [[AssignmentChar](#TAssignmentChar) [TemplateBody](#TConstantExpression)] {","

 [Identifier](#TIdentifier) [[AssignmentChar](#TAssignmentChar) [TemplateBody](#TConstantExpression)]}

#### A.1.6.1.13 Friend module definitions

FriendModuleDef ::= "friend" "module" [IdentifierList](#TIdentifierList) [[SemiColon](#TSemiColon)]

### A.1.6.2 Module control function

ModuleControlDef ::= [ControlKeyword](#TControlKeyword) [StatementBlock](#TStatementBlock)

ControlKeyword ::= "control"

### A.1.6.3 Local definitions

#### A.1.6.3.1 Variable instantiation

VarInstance ::= [VarKeyword](#TVarKeyword) (([([LazyModifier](#TLazyModifier) | [FuzzyModifier](#TFuzzyModifier)) [DeterministicModifier] ]

 [Type](#TType) [VarList](#TVarList)) |

 (([TemplateKeyword](#TTemplateKeyword) | [RestrictedTemplate](#TRestrictedTemplate))

 [([LazyModifier](#TLazyModifier) | [FuzzyModifier](#TFuzzyModifier)) ) [DeterministicModifier] ]

 [Type](#TType) [TempVarList](#TTempVarList)))

VarList ::= [SingleVarInstance](#TSingleVarInstance) {"," [SingleVarInstance](#TSingleVarInstance)}

SingleVarInstance ::= [Identifier](#TIdentifier) [[ArrayDef](#TArrayDef)] [[AssignmentChar](#TAssignmentChar) [Expression](#TExpression)]

VarKeyword ::= "var"

TempVarList ::= [SingleTempVarInstance](#TSingleTempVarInstance) {"," [SingleTempVarInstance](#TSingleTempVarInstance)}

SingleTempVarInstance ::= [Identifier](#TIdentifier) [[ArrayDef](#TArrayDef)] [[AssignmentChar](#TAssignmentChar) [TemplateBody](#TTemplateBody)]

ValueRef ::= [Identifier](#TIdentifier) [[ExtendedFieldReference](#TExtendedFieldReference)]

#### A.1.6.3.2 Timer instantiation

TimerInstance ::= [TimerKeyword](#TTimerKeyword) [VarList](#TVarList)

TimerKeyword ::= "timer"

ArrayIdentifierRef ::= [Identifier](#TIdentifier) {[ArrayOrBitRef](#TArrayOrBitRef)}

### A.1.6.4 Operations

#### A.1.6.4.1 Component operations

ConfigurationStatements ::= [ConnectStatement](#TConnectStatement) |

 [MapStatement](#TMapStatement) |

 [DisconnectStatement](#TDisconnectStatement) |

 [UnmapStatement](#TUnmapStatement) |

 [ [NoDefaultModifier](#TNoDefaultModifier) ] [DoneStatement](#TDoneStatement) |

 [ [NoDefaultModifier](#TNoDefaultModifier) ] [KilledStatement](#TKilledStatement) |

 [StartTCStatement](#TStartTCStatement) |

 [StopTCStatement](#TStopTCStatement) |

 [KillTCStatement](#TKillTCStatement) |

 SetEncodeStatement

ConfigurationOps ::= [CreateOp](#TCreateOp) |

 [SelfOp](#TSelfOp) |

 [SystemKeyword](#TSystemKeyword) |

 [MTCKeyword](#TMTCKeyword) |

 [RunningOp](#TRunningOp) |

 [AliveOp](#TAliveOp)

CreateOp ::= [ComponentType](#TComponentType) [Dot](#TDot) [CreateKeyword](#TCreateKeyword) ["(" ([SingleExpression](#TSingleExpression) |

 [Minus](#TMinus)) ["," [SingleExpression](#TSingleExpression)] ")"] [[AliveKeyword](#TAliveKeyword)]

SelfOp ::= "self"

DoneStatement ::= [ComponentOrAny](#TComponentOrAny) [Dot](#TDot) [DoneKeyword](#TDoneKeyword) [ [PortRedirectSymbol](#TPortRedirectSymbol)

 [ [ValueStoreSpec](#TValueStoreSpec) ] [ [IndexSpec](#TIndexSpec) ] ]

/\*STATIC SEMANTICS – If [*PortRedirectSymbol*](#TPortRedirectSymbol) is present, at least one of *ValueStoreSpec* and [*IndexSpec*](#TIndexSpec) shall be present\*/

ComponentOrAny ::= [ObjectReference](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C05PWRTCH%5CCR7707-v5.docx#TObjectReference) |

 ([AnyKeyword](#TAnyKeyword) ([ComponentKeyword](#TComponentKeyword) | [FromKeyword](#TFromKeyword) ValueRef)) |

 ([AllKeyword](#TAllKeyword) [ComponentKeyword](#TComponentKeyword))

ValueStoreSpec ::= ValueKeyword ValueRef

IndexAssignment ::= [PortRedirectSymbol](#TPortRedirectSymbol) [IndexSpec](#TIndexSpec)

IndexSpec ::= [IndexModifier](#TIndexModifier) [ValueStoreSpec](#TValueStoreSpec)

KilledStatement ::= [ComponentOrAny](#TComponentOrAny) [Dot](#TDot) [KilledKeyword](#TKilledKeyword) [ [PortRedirectSymbol](#TPortRedirectSymbol)

 [ [ValueStoreSpec](#TValueStoreSpec) ] [ [IndexSpec](#TIndexSpec)] ]

/\*STATIC SEMANTICS – If [*PortRedirectSymbol*](#TPortRedirectSymbol) is present, at least one of *ValueStoreSpec* and [*IndexSpec*](#TIndexSpec) shall be present\*/

DoneKeyword ::= "done"

KilledKeyword ::= "killed"

RunningOp ::= [ComponentOrAny](#TComponentOrAny) [Dot](#TDot) [RunningKeyword](#TRunningKeyword) [[IndexAssignment](#TIndexAssignment)]

RunningKeyword ::= "running"

AliveOp ::= [ComponentOrAny](#TComponentOrAny) [Dot](#TDot) [AliveKeyword](#TAliveKeyword) [[IndexAssignment](#TIndexAssignment)]

CreateKeyword ::= "create"

AliveKeyword ::= "alive"

ConnectStatement ::= [ConnectKeyword](#TConnectKeyword) [SingleConnectionSpec](#TSingleConnectionSpec)

ConnectKeyword ::= "connect"

SingleConnectionSpec ::= "(" [PortRef](#TPortRef) "," [PortRef](#TPortRef) ")"

PortRef ::= [ComponentRef](#TComponentRef) [Colon](#TColon) [ArrayIdentifierRef](#TArrayIdentifierRef)

ComponentRef ::= [ObjectReference](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C05PWRTCH%5CCR7707-v5.docx#TObjectReference) |

 [SystemKeyword](#TSystemKeyword) |

 [SelfOp](#TSelfOp) |

 [MTCKeyword](#TMTCKeyword)

DisconnectStatement ::= [DisconnectKeyword](#TDisconnectKeyword) [[SingleConnectionSpec](#TSingleConnectionSpec) |

 [AllConnectionsSpec](#TAllConnectionsSpec) |

 [AllPortsSpec](#TAllPortsSpec) |

 [AllCompsAllPortsSpec](#TAllCompsAllPortsSpec)

 ]

AllConnectionsSpec ::= "(" [PortRef](#TPortRef) ")"

AllPortsSpec ::= "(" [ComponentRef](#TComponentRef) ":" [AllKeyword](#TAllKeyword) [PortKeyword](#TPortKeyword) ")"

AllCompsAllPortsSpec ::= "(" [AllKeyword](#TAllKeyword) [ComponentKeyword](#TComponentKeyword) ":" [AllKeyword](#TAllKeyword)

 [PortKeyword](#TPortKeyword) ")"

DisconnectKeyword ::= "disconnect"

MapStatement ::= [MapKeyword](#TMapKeyword) [SingleConnectionSpec](#TSingleConnectionSpec) [[ParamClause](#TParamClause)]

ParamClause ::= [ParamKeyword](#TParamKeyword) [ActualParList](#TActualParList)

MapKeyword ::= "map"

UnmapStatement ::= [UnmapKeyword](#TUnmapKeyword) [[SingleConnectionSpec](#TSingleConnectionSpec) [[ParamClause](#TParamClause)] |

 [AllConnectionsSpec](#TAllConnectionsSpec) [[ParamClause](#TParamClause)] |

 [AllPortsSpec](#TAllPortsSpec) |

 [AllCompsAllPortsSpec](#TAllCompsAllPortsSpec)

 ]

UnmapKeyword ::= "unmap"

StartTCStatement ::= [ObjectReference](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C05PWRTCH%5CCR7707-v5.docx#TObjectReference) [Dot](#TDot) [StartKeyword](#TStartKeyword)

 "(" ([FunctionInstance](#TFunctionInstance) | [AltstepInstance](#TAltstepInstance)) ")"

StartKeyword ::= "start"

StopTCStatement ::= [StopKeyword](#TStopKeyword) | ([ComponentReferenceOrLiteral](#TComponentReferenceOrLiteral) | [AllKeyword](#TAllKeyword)

 [ComponentKeyword](#TComponentKeyword)) [Dot](#TDot) [StopKeyword](#TStopKeyword)

ComponentReferenceOrLiteral ::= [ObjectReference](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C05PWRTCH%5CCR7707-v5.docx#TObjectReference) |

 [MTCKeyword](#TMTCKeyword) |

 [SelfOp](#TSelfOp)

KillTCStatement ::= [KillKeyword](#TKillKeyword) | (([ComponentReferenceOrLiteral](#TComponentReferenceOrLiteral) |

 [AllKeyword](#TAllKeyword) [ComponentKeyword](#TComponentKeyword)) [Dot](#TDot) [KillKeyword](#TKillKeyword))

[ObjectReference](file:///C%3A%5C%5CUsers%5C%5Cethgry%5C%5CAppData%5C%5CLocal%5C%5CMicrosoft%5C%5CWindows%5C%5CTemporary%20Internet%20Files%5C%5CContent.IE5%5C%5C05PWRTCH%5C%5CCR7707-v5.docx%22%20%5Cl%20%22TObjectReference) ::= ValueRef | [FunctionInstance](#TFunctionInstance)

KillKeyword ::= "kill"

SetEncodeStatement ::= ( [SingleExpression](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C08RUYH79%5CCR7714-v1.docx#TSingleExpression) | ( [AllKeyword](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C08RUYH79%5CCR7714-v1.docx#TAllKeyword) [PortKeyword](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C08RUYH79%5CCR7714-v1.docx#TPortKeyword) ) | [SelfOp](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C08RUYH79%5CCR7714-v1.docx#TSelfOp) ) | "."
 SetEncodeKeyword "(" [Type](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C08RUYH79%5CCR7714-v1.docx#TType) "," [SingleExpression](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C08RUYH79%5CCR7714-v1.docx#TSingleExpression) ")"

SetEncodeKeyword ::= "setencode"

#### A.1.6.4.2 Port operations

CommunicationStatements ::= [SendStatement](#TSendStatement) |

 [CallStatement](#TCallStatement) |

 [ReplyStatement](#TReplyStatement) |

 [RaiseStatement](#TRaiseStatement) |

 [ [NoDefaultModifier](#TNoDefaultModifier) ] [ReceivingCommunicationStatement](#TReceivingCommunicationStatements) |

 [ClearStatement](#TClearStatement) |

 [StartStatement](#TStartStatement) |

 [StopStatement](#TStopStatement) |

 [HaltStatement](#THaltStatement) |

 [CheckStateStatement](#TCheckStateStatement)

ReceivingCommunicationStatement ::= [ReceiveStatement](#TReceiveStatement) |

 [TriggerStatement](#TTriggerStatement) |

 [GetCallStatement](#TGetCallStatement) |

 [GetReplyStatement](#TGetReplyStatement) |

 [CatchStatement](#TCatchStatement)

SendStatement ::= [ObjectReference](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C05PWRTCH%5CCR7707-v5.docx#TObjectReference) [Dot](#TDot) [PortSendOp](#TPortSendOp)

PortSendOp ::= [SendOpKeyword](#TSendOpKeyword) "(" [TemplateInstance](#TTemplateInstance)")" [[ToClause](#TToClause)]

SendOpKeyword ::= "send"

ToClause ::= [ToKeyword](#TToKeyword) ([TemplateInstance](#TTemplateInstance)|

 [AddressRefList](#TAddressRefList) |

 [AllKeyword](#TAllKeyword) [ComponentKeyword](#TComponentKeyword)

 )

AddressRefList ::= "(" [TemplateInstance](#TTemplateInstance){"," [TemplateInstance](#TTemplateInstance)} ")"

ToKeyword ::= "to"

CallStatement ::= [ObjectReference](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C05PWRTCH%5CCR7707-v5.docx#TObjectReference) [Dot](#TDot) [PortCallOp](#TPortCallOp) [[PortCallBody](#TPortCallBody)]

PortCallOp ::= [CallOpKeyword](#TCallOpKeyword) "(" [CallParameters](#TCallParameters) ")" [[ToClause](#TToClause)]

CallOpKeyword ::= "call"

CallParameters ::= [TemplateInstance](#TTemplateInstance) ["," [CallTimerValue](#TCallTimerValue)]

CallTimerValue ::= [Expression](#TExpression) | [NowaitKeyword](#TNowaitKeyword)

NowaitKeyword ::= "nowait"

PortCallBody ::= "{" [CallBodyStatementList](#TCallBodyStatementList) "}"

CallBodyStatementList ::= {[CallBodyStatement](#TCallBodyStatement) [[SemiColon](#TSemiColon)]}+

CallBodyStatement ::= [CallBodyGuard](#TCallBodyGuard) [StatementBlock](#TStatementBlock)

CallBodyGuard ::= [AltGuardChar](#TAltGuardChar) [CallBodyOps](#TCallBodyOps)

CallBodyOps ::= [GetReplyStatement](#TGetReplyStatement) | [CatchStatement](#TCatchStatement)

ReplyStatement ::= [ObjectReference](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C05PWRTCH%5CCR7707-v5.docx#TObjectReference) [Dot](#TDot) [PortReplyOp](#TPortReplyOp)

PortReplyOp ::= [ReplyKeyword](#TReplyKeyword) "(" [TemplateInstance](#TTemplateInstance) [[ReplyValue](#TReplyValue)] ")" [[ToClause](#TToClause)]

ReplyKeyword ::= "reply"

ReplyValue ::= [ValueKeyword](#TValueKeyword) [TemplateBody](#TTemplateBody)

/\* STATIC SEMANTICS - TemplateBody shall be type compatible with the return type. It shall evaluate to a value or template (literal or template instance) conforming to the template(value) restriction. \*/

RaiseStatement ::= [ObjectReference](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C05PWRTCH%5CCR7707-v5.docx#TObjectReference) [Dot](#TDot) [PortRaiseOp](#TPortRaiseOp)

PortRaiseOp ::= [RaiseKeyword](#TRaiseKeyword) "(" [Signature](#TSignature) "," [TemplateInstance](#TTemplateInstance)")"

 [[ToClause](#TToClause)]

RaiseKeyword ::= "raise"

NoDefaultModifier ::= "@nodefault"

ReceiveStatement ::= [PortOrAny](#TPortOrAny) [Dot](#TDot) [PortReceiveOp](#TPortReceiveOp)

PortOrAny ::= [ObjectReference](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C05PWRTCH%5CCR7707-v5.docx#TObjectReference) | ([AnyKeyword](#TAnyKeyword) ([PortKeyword](#TPortKeyword) | [FromKeyword](#TFromKeyword) ValueRef))

PortReceiveOp ::= [ReceiveOpKeyword](#TReceiveOpKeyword) ["("[TemplateInstance](#TTemplateInstance)")"] [[FromClause](#TFromClause)] [[PortRedirect](#TPortRedirect)]

ReceiveOpKeyword ::= "receive"

FromClause ::= [FromKeyword](#TFromKeyword) ([TemplateInstance](#TTemplateInstance) |

 [AddressRefList](#TAddressRefList) |

 [AnyKeyword](#TAnyKeyword) [ComponentKeyword](#TComponentKeyword)

 )

FromKeyword ::= "from"

PortRedirect ::= [PortRedirectSymbol](#TPortRedirectSymbol) (([ValueSpec](#TValueSpec) [[SenderSpec](#TSenderSpec)] [[IndexSpec](#TIndexSpec)]) |

 ([SenderSpec](#TSenderSpec) [[IndexSpec](#TIndexSpec)]) |

 [IndexSpec](#TIndexSpec)

 )

PortRedirectSymbol ::= "->"

ValueSpec ::= [ValueKeyword](#TValueKeyword) (ValueRef | ("(" [SingleValueSpec](#TSingleValueSpec) {"," [SingleValueSpec](#TSingleValueSpec)} ")"))

SingleValueSpec ::= ValueRef [[AssignmentChar](#TAssignmentChar) [ DecodedModifier ["(" [Expression] ")"] ]

 [FieldReference](#TFieldReference) [ExtendedFieldReference](#TExtendedFieldReference)]

/\*STATIC SEMANTICS – FieldReference shall not be ParRef and ExtendedFieldReference shall not be TypeDefIdentifier\*/

ValueKeyword ::= "value"

SenderSpec ::= [SenderKeyword](#TSenderKeyword) ValueRef

SenderKeyword ::= "sender"

TriggerStatement ::= [PortOrAny](#TPortOrAny) [Dot](#TDot) [PortTriggerOp](#TPortTriggerOp)

PortTriggerOp ::= [TriggerOpKeyword](#TTriggerOpKeyword) ["(" [TemplateInstance](#TInLineTemplate)  ")"] [[FromClause](#TFromClause)]

 [[PortRedirect](#TPortRedirect)]

TriggerOpKeyword ::= "trigger"

GetCallStatement ::= [PortOrAny](#TPortOrAny) [Dot](#TDot) [PortGetCallOp](#TPortGetCallOp)

PortGetCallOp ::= [GetCallOpKeyword](#TGetCallOpKeyword) ["(" [TemplateInstance](#TTemplateInstance) ")"] [[FromClause](#TFromClause)]

 [[PortRedirectWithParam](#TPortRedirectWithParam)]

GetCallOpKeyword ::= "getcall"

PortRedirectWithParam ::= [PortRedirectSymbol](#TPortRedirectSymbol) [RedirectWithParamSpec](#TRedirectWithParamSpec)

RedirectWithParamSpec ::= ([ParamSpec](#TParamSpec) [[SenderSpec](#TSenderSpec)] [[IndexSpec](#TIndexSpec)]) |

 ([SenderSpec](#TSenderSpec) [[IndexSpec](#TIndexSpec)]) |

 [IndexSpec](#TIndexSpec)

ParamSpec ::= [ParamKeyword](#TParamKeyword) [ParamAssignmentList](#TParamAssignmentList)

ParamKeyword ::= "param"

ParamAssignmentList ::= "(" ([AssignmentList](#TAssignmentList) | [VariableList](#TVariableList)) ")"

AssignmentList ::= [VariableAssignment](#TVariableAssignment) {"," [VariableAssignment](#TVariableAssignment)}

VariableAssignment ::= ValueRef [AssignmentChar](#TAssignmentChar) [ [DecodedModifier](#TDecodedModifier) ["(" [Expression](#TExpression)] ")"]

 [Identifier](#TIdentifier)

VariableList ::= [VariableEntry](#TVariableEntry) {"," [VariableEntry](#TVariableEntry)}

VariableEntry ::= ValueRef | [Minus](#TMinus)

GetReplyStatement ::= [PortOrAny](#TPortOrAny) [Dot](#TDot) [PortGetReplyOp](#TPortGetReplyOp)

PortGetReplyOp ::= [GetReplyOpKeyword](#TGetReplyOpKeyword) ["(" [TemplateInstance](#TTemplateInstance) [[ValueMatchSpec](#TValueMatchSpec)]

 ")"] [[FromClause](#TFromClause)] [[PortRedirectWithValueAndParam](#TPortRedirectWithValueAndParam)]

PortRedirectWithValueAndParam ::= [PortRedirectSymbol](#TPortRedirectSymbol) [RedirectWithValueAndParamSpec](#TRedirectWithValueAndParamSpec)

RedirectWithValueAndParamSpec ::= ([ValueSpec](#TValueSpec) [[ParamSpec](#TParamSpec)] [[SenderSpec](#TSenderSpec)]

 [[IndexSpec](#TIndexSpec)]) | [RedirectWithParamSpec](#TRedirectWithParamSpec)

GetReplyOpKeyword ::= "getreply"

ValueMatchSpec ::= [ValueKeyword](#TValueKeyword) [TemplateInstance](#TTemplateInstance)

CheckStatement ::= [PortOrAny](#TPortOrAny) [Dot](#TDot) [PortCheckOp](#TPortCheckOp)

PortCheckOp ::= [CheckOpKeyword](#TCheckOpKeyword) ["(" [CheckParameter](#TCheckParameter) ")"]

CheckOpKeyword ::= "check"

CheckParameter ::= [CheckPortOpsPresent](#TCheckPortOpsPresent) |

 [FromClausePresent](#TFromClausePresent) |

 [RedirectPresent](#TRedirectPresent)

FromClausePresent ::= [FromClause](#TFromClause) [[PortRedirectSymbol](#TPortRedirectSymbol) (([SenderSpec](#TSenderSpec)

 [[IndexSpec](#TIndexSpec)]) |

 [IndexSpec](#TIndexSpec))]

RedirectPresent ::= [PortRedirectSymbol](#TPortRedirectSymbol) (([SenderSpec](#TSenderSpec) [[IndexSpec](#TIndexSpec)]) |

 [IndexSpec](#TIndexSpec))

CheckPortOpsPresent ::= [PortReceiveOp](#TPortReceiveOp) |

 [PortGetCallOp](#TPortGetCallOp) |

 [PortGetReplyOp](#TPortGetReplyOp) |

 [PortCatchOp](#TPortCatchOp)

CatchStatement ::= [PortOrAny](#TPortOrAny) [Dot](#TDot) [PortCatchOp](#TPortCatchOp)

PortCatchOp ::= [CatchOpKeyword](#TCatchOpKeyword) ["(" [CatchOpParameter](#TCatchOpParameter) ")"] [[FromClause](#TFromClause)] [[PortRedirect](#TPortRedirect)]

CatchOpKeyword ::= "catch"

CatchOpParameter ::= [Signature](#TSignature) [ "," [TemplateInstance](#TTemplateInstance) ] | [TimeoutKeyword](#TTimeoutKeyword)

ClearStatement ::= [PortOrAll](#TPortOrAll) [Dot](#TDot) [ClearOpKeyword](#TClearOpKeyword)

PortOrAll ::= [ObjectReference](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C05PWRTCH%5CCR7707-v5.docx#TObjectReference) | [AllKeyword](#TAllKeyword) [PortKeyword](#TPortKeyword)

ClearOpKeyword ::= "clear"

StartStatement ::= [PortOrAll](#TPortOrAll) [Dot](#TDot) [StartKeyword](#TStartKeyword)

StopStatement ::= [PortOrAll](#TPortOrAll) [Dot](#TDot) [StopKeyword](#TStopKeyword)

StopKeyword ::= "stop"

HaltStatement ::= [PortOrAll](#TPortOrAll) [Dot](#TDot) [HaltKeyword](#THaltKeyword)

HaltKeyword ::= "halt"

AnyKeyword ::= "any"

CheckStateStatement ::= [PortOrAllAny](#TPortOrAllAny) [Dot](#TDot) [CheckStateKeyword](#TCheckStateKeyword) "(" [SingleExpression](#TSingleExpression)

 ")"

PortOrAllAny ::= [PortOrAll](#TPortOrAll) | [AnyKeyword](#TAnyKeyword) [PortKeyword](#TPortKeyword)

CheckStateKeyword ::= "checkstate"

#### A.1.6.4.3 Timer operations

TimerStatements ::= [StartTimerStatement](#TStartTimerStatement) |

 [StopTimerStatement](#TStopTimerStatement) |

 [ [NoDefaultModifier](#TNoDefaultModifier) ] [TimeoutStatement](#TTimeoutStatement)

TimerOps ::= [ReadTimerOp](#TReadTimerOp) | [RunningTimerOp](#TRunningTimerOp)

StartTimerStatement ::= [ObjectReference](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C05PWRTCH%5CCR7707-v5.docx#TObjectReference) [Dot](#TDot) [StartKeyword](#TStartKeyword) ["(" [Expression](#TExpression) ")"]

StopTimerStatement ::= [TimerRefOrAll](#TTimerRefOrAll) [Dot](#TDot) [StopKeyword](#TStopKeyword)

TimerRefOrAll ::= [ObjectReference](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C05PWRTCH%5CCR7707-v5.docx#TObjectReference) | [AllKeyword](#TAllKeyword) [TimerKeyword](#TTimerKeyword)

ReadTimerOp ::= [ObjectReference](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C05PWRTCH%5CCR7707-v5.docx#TObjectReference) [Dot](#TDot) [ReadKeyword](#TReadKeyword)

ReadKeyword ::= "read"

RunningTimerOp ::= [TimerRefOrAny](#TTimerRefOrAny) [Dot](#TDot) [RunningKeyword](#TRunningKeyword) [[IndexAssignment](#TIndexAssignment)]

TimeoutStatement ::= [TimerRefOrAny](#TTimerRefOrAny) [Dot](#TDot) [TimeoutKeyword](#TTimeoutKeyword) [[IndexAssignment](#TIndexAssignment)]

TimerRefOrAny ::= [ObjectReference](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C05PWRTCH%5CCR7707-v5.docx#TObjectReference) |

 ([AnyKeyword](#TAnyKeyword) [TimerKeyword](#TTimerKeyword)) |

 ([AnyKeyword](#TAnyKeyword) [FromKeyword](#TFromKeyword) [Identifier](#TIdentifier))

TimeoutKeyword ::= "timeout"

#### A.1.6.4.4 Testcase operation

TestcaseOperation ::= [TestcaseKeyword](#TTestcaseKeyword) "." [StopKeyword](#TStopKeyword) ["(" { [LogItem](#TLogItem) [","] } ")"]

### A.1.6.5 Type

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

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

 [BooleanKeyword](#TBooleanKeyword) |

 [CharStringKeyword](#TCharStringKeyword) |

 [UniversalCharString](#TUniversalCharString) |

 [IntegerKeyword](#TIntegerKeyword) |

 [OctetStringKeyword](#TOctetStringKeyword) |

 [HexStringKeyword](#THexStringKeyword) |

 [VerdictTypeKeyword](#TVerdictTypeKeyword) |

 [FloatKeyword](#TFloatKeyword) |

 [AddressKeyword](#TAddressKeyword) |

 [DefaultKeyword](#TDefaultKeyword) |

 [AnyTypeKeyword](#TAnyTypeKeyword) |

 TimerKeyword

BitStringKeyword ::= "bitstring"

BooleanKeyword ::= "boolean"

IntegerKeyword ::= "integer"

OctetStringKeyword ::= "octetstring"

HexStringKeyword ::= "hexstring"

VerdictTypeKeyword ::= "verdicttype"

FloatKeyword ::= "float"

AddressKeyword ::= "address"

DefaultKeyword ::= "default"

AnyTypeKeyword ::= "anytype"

CharStringKeyword ::= "charstring"

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

UniversalKeyword ::= "universal"

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

TypeReference ::= [ExtendedIdentifier](#TExtendedIdentifier)

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

ExtendedTypeFieldReference ::= {([Dot](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C0EG46CRK%5CCR7496-v1.docx#TDot) ([Identifier](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C0EG46CRK%5CCR7496-v1.docx#TIdentifier) | [PredefinedType](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C0EG46CRK%5CCR7496-v1.docx#TPredefinedType))) |

 ("[" [Minus](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C0EG46CRK%5CCR7496-v1.docx#TMinus) "]") }+

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

### A.1.6.6 Value

Value ::= [PredefinedValue](#TPredefinedValue) | [ReferencedValue](#TReferencedValue)

PredefinedValue ::= [Bstring](#TBstring) |

 [BooleanValue](#TBooleanValue) |

 [CharStringValue](#TCharStringValue) |

 [Number](#TNumber) | /\* [IntegerValue](#TIntegerValue) \*/

 [Ostring](#TOstring) |

 [Hstring](#THstring) |

 [VerdictTypeValue](#TVerdictTypeValue) |

 [FloatValue](#TFloatValue) |

 [AddressValue](#TAddressValue) |

 [OmitKeyword](#TOmitKeyword)

BooleanValue ::= "true" | "false"

VerdictTypeValue ::= "pass" |

 "fail" |

 "inconc" |

 "none" |

 "error"

CharStringValue ::= [Cstring](#TCstring) | [Quadruple](#TQuadruple) | [USIlikeNotation](#TUSIlikeNotation)

Quadruple ::= [CharKeyword](#TCharKeyword) "(" [Number](#TNumber) "," [Number](#TNumber) "," [Number](#TNumber) "," [Number](#TNumber) ")"

USIlikeNotation ::= [CharKeyword](#TCharKeyword) "(" [UIDlike](#TUIDlike) { "," [UIDlike](#TUIDlike) } ")"

UIDlike ::= ("U"|"u") {"+"} {[Hex](#THex)}#(1,8)

CharKeyword ::= "char"

FloatValue ::= [FloatDotNotation](#TFloatDotNotation) |

 [FloatENotation](#TFloatENotation) |

 [NaNKeyword](#TNaNKeyword)

NaNKeyword ::= "not\_a\_number"

FloatDotNotation ::= [Number](#TNumber) [Dot](#TDot) [DecimalNumber](#TDecimalNumber)

FloatENotation ::= [Number](#TNumber) [[Dot](#TDot) [DecimalNumber](#TDecimalNumber)] [Exponential](#TExponential) [[Minus](#TMinus)] [Number](#TNumber)

Exponential ::= "E"

ReferencedValue ::= ([ExtendedIdentifier](#TExtendedIdentifier) [[ExtendedFieldReference](#TExtendedFieldReference)] ) | ReferencedEnumValue

/\* STATIC SEMANTICS – The second option is used only for referencing enumerated values, in all other cases, the first option is used.

ReferencedEnumValue ::= [ReferencedType [Dot](#TDot)] [Identifier](#TIdentifier) [[ExtendedEnumReference](#TExtendedEnumReference)]

/\*\* STATIC Semantics: ExtendedEnumReference shall be present if and only if Identifier refers to an enumerated value with an attached value list \*/

ExtendedEnumReference ::= "(" [IntegerValue](#TIntegerValue) ")"

Number ::= ([NonZeroNum](#TNonZeroNum) {[Num](#TNum)}) | "0"

NonZeroNum ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

DecimalNumber ::= { [Num](#TNum) }+

Num ::= "0" | [NonZeroNum](#TNonZeroNum)

Bstring ::= "'" { [Bin](#TBin) | [BinSpace](#TBinSpace) } "'" "B"

Bin ::= "0" | "1"

Hstring ::= "'" { [Hex](#THex) | [BinSpace](#TBinSpace) } "'" "H"

Hex ::= [Num](#TNum) | "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" |

 "d" | "e" | "f"

Ostring ::= "'" { [Oct](#TOct) | [BinSpace](#TBinSpace) } "'" "O"

Oct ::= [Hex](#THex) [Hex](#THex)

Cstring ::= """ {[Char](#TChar)} """

Char ::= /\* REFERENCE - A character defined by the relevant CharacterString type. For charstring a character from the character set defined in ITU-T T.50. For universal charstring a character from any character set defined in ISO/IEC 10646 \*/

Identifier ::= [Alpha](#TAlpha) {[AlphaNum](#TAlphaNum) | [Underscore](#TUnderscore)}

Alpha ::= [UpperAlpha](#TUpperAlpha) | [LowerAlpha](#TLowerAlpha)

AlphaNum ::= [Alpha](#TAlpha) | [Num](#TNum)

UpperAlpha ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |

 "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |

 "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"

LowerAlpha ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |

 "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |

 "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

ExtendedAlphaNum ::= /\* REFERENCE - A graphical character from the BASIC LATIN or from the LATIN-1 SUPPLEMENT character sets defined in ISO/IEC 10646 (characters from char (0,0,0,32) to char (0,0,0,126), from char (0,0,0,161) to char (0,0,0,172) and from char (0,0,0,174) to char (0,0,0,255) \*/

FreeText ::= """ {[ExtendedAlphaNum](#TExtendedAlphaNum)} """

AddressValue ::= "null"

OmitKeyword ::= "omit"

 BinSpace ::= " " | "\" [NLChar](#TNLChar)

 NLChar ::= /\* REFERENCE - Any sequence of newline characters that constitute a newline by using the following C0 control characters: LF(10), VT(11), FF(12), CR(13) (see Recommendation ITU‑T T.50 [4]) (jointly called newline characters, see clause A.1.5.1) from the character set defined in Recommendation ITU‑T T.50 [4].\*/

### A.1.6.7 Parameterization

InParKeyword ::= "in"

OutParKeyword ::= "out"

InOutParKeyword ::= "inout"

FormalValuePar ::= [[InParKeyword](#TInParKeyword) | [InOutParKeyword](#TInOutParKeyword) | [OutParKeyword](#TOutParKeyword) )]

 [([LazyModifier](#TLazyModifier) | [FuzzyModifier](#TFuzzyModifier)) [DeterministicModifier] ]

 [Type](#TType) [Identifier](#TIdentifier) [[ArrayDef](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C0EG46CRK%5CCR7496-v1.docx#TArrayDef)] [":=" ([Expression](#TExpression) | [Minus](#TMinus))]

FormalTemplatePar ::= [([InParKeyword](#TInParKeyword) | [OutParKeyword](#TOutParKeyword) | [InOutParKeyword](#TInOutParKeyword))]

 ([TemplateKeyword](#TTemplateKeyword) | [RestrictedTemplate](#TRestrictedTemplate))

 [([LazyModifier](#TLazyModifier) | [FuzzyModifier](#TFuzzyModifier)) [DeterministicModifier] ]

 [Type](#TType) [Identifier](#TIdentifier) [[ArrayDef](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C0EG46CRK%5CCR7496-v1.docx#TArrayDef)] [":=" ([TemplateInstance](#TTemplateInstance) | [Minus](#TMinus)) ]

RestrictedTemplate ::= [OmitKeyword](#TOmitKeyword) | ([TemplateKeyword](#TTemplateKeyword) [TemplateRestriction](#TTemplateRestriction))

TemplateRestriction ::= "(" ([OmitKeyword](#TOmitKeyword) |

 [ValueKeyword](#TValueKeyword) |

 [PresentKeyword](#TPresentKeyword)

 ) ")"

### A.1.6.8 Statements

#### A.1.6.8.1 With statement

WithStatement ::= [WithKeyword](#TWithKeyword) [WithAttribList](#TWithAttribList)

WithKeyword ::= "with"

WithAttribList ::= "{" [MultiWithAttrib](#TMultiWithAttrib) "}"

MultiWithAttrib ::= {[SingleWithAttrib](#TSingleWithAttrib) [[SemiColon](#TSemiColon)]}

SingleWithAttrib ::= StandardAttribute |

 VariantAttribute

StandardAttribute ::= [AttribKeyword](#TAttribKeyword) [[OverrideKeyword](#TOverrideKeyword) | LocalModifier] [[AttribQualifier](#TAttribQualifier)]

 [FreeText](#TFreeText)

 VariantAttribute ::= [VariantKeyword](#TVariantKeyword) [( [OverrideKeyword](#TOverrideKeyword) | LocalModifier )]

 [[AttribQualifier](#TAttribQualifier)] [ [RelatedEncoding](#TRelatedEncoding) "." ] [FreeText](#TFreeText)

 RelatedEncoding ::= [FreeText](#TFreeText) | ( "{" [FreeText](#TFreeText) { "," [FreeText](#TFreeText) } "}" )

AttribKeyword ::= [EncodeKeyword](#TEncodeKeyword) |

 [DisplayKeyword](#TDisplayKeyword) |

 [ExtensionKeyword](#TExtensionKeyword) |

 [OptionalKeyword](#TOptionalKeyword)

EncodeKeyword ::= "encode"

VariantKeyword ::= "variant"

DisplayKeyword ::= "display"

ExtensionKeyword ::= "extension"

OverrideKeyword ::= "override"

LocalModifier ::= "@local"

AttribQualifier ::= "(" [DefOrFieldRefList](#TDefOrFieldRefList) ")"

DefOrFieldRefList ::= [DefOrFieldRef](#TDefOrFieldRef) {"," [DefOrFieldRef](#TDefOrFieldRef)}

DefOrFieldRef ::= [QualifiedIdentifier](#TQualifiedIdentifier) |

 (([FieldReference](#TFieldReference) | "[" [Minus](#TMinus) "]") [[ExtendedFieldOrTypeReference](#TExtendedFieldReference)]) |

 [AllRef](#TAllRef)

QualifiedIdentifier ::= {[Identifier](#TIdentifier) [Dot](#TDot)} [Identifier](#TIdentifier)

ExtendedFieldOrTypeReference ::= {([Dot](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C0EG46CRK%5CCR7496-v1.docx#TDot) ([Identifier](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C0EG46CRK%5CCR7496-v1.docx#TIdentifier) | [PredefinedType](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C0EG46CRK%5CCR7496-v1.docx#TPredefinedType))) |

 [ArrayOrBitRef](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C0EG46CRK%5CCR7496-v1.docx#TArrayOrBitRef) | ("[" [Minus](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C0EG46CRK%5CCR7496-v1.docx#TMinus) "]") }+

/\* STATIC SEMANTIC - The Identifier refers to a type definition if the type of the VarInstance or ReferencedValue in which the ExtendedFieldOrReference is used is anytype. ArrayOrBitRef shall be used when referencing elements of values or arrays. The square brackets with dash shall be used when referencing inner types of a record of, set of or array type. \*/

AllRef ::= ([GroupKeyword](#TGroupKeyword) [AllKeyword](#TAllKeyword) [[ExceptKeyword](#TExceptKeyword) "{" [QualifiedIdentifierList](#TQualifiedIdentifierList)

 "}"]) | (([TypeDefKeyword](#TTypeDefKeyword) |

 [TemplateKeyword](#TTemplateKeyword) |

 [ConstKeyword](#TConstKeyword) |

 [AltstepKeyword](#TAltstepKeyword) |

 [TestcaseKeyword](#TTestcaseKeyword) |

 [FunctionKeyword](#TFunctionKeyword) |

 [SignatureKeyword](#TSignatureKeyword) |

 [ModuleParKeyword](#TModuleParKeyword)

 ) [AllKeyword](#TAllKeyword) [[ExceptKeyword](#TExceptKeyword)

 "{" [IdentifierList](#TIdentifierList)

 "}"])

#### A.1.6.8.2 Behaviour statements

BehaviourStatements ::= [TestcaseInstance](#TTestcaseInstance) |

 [FunctionInstance](#TFunctionInstance) |

 [ReturnStatement](#TReturnStatement) |

 [AltConstruct](#TAltConstruct) |

 [InterleavedConstruct](#TInterleavedConstruct) |

 [LabelStatement](#TLabelStatement) |

 [GotoStatement](#TGotoStatement) |

 [RepeatStatement](#TRepeatStatement) |

 [DeactivateStatement](#TDeactivateStatement) |

 [AltstepInstance](#TAltstepInstance) |

 [ActivateOp](#TActivateOp) |

 [BreakStatement](#TBreakStatement) |

 [ContinueStatement](#TContinueStatement)

SetLocalVerdict ::= [SetVerdictKeyword](#TSetVerdictKeyword) "(" [SingleExpression](#TSingleExpression) {"," [LogItem](#TLogItem)}

 ")"

SetVerdictKeyword ::= "setverdict"

GetLocalVerdict ::= "getverdict"

SUTStatements ::= [ActionKeyword](#TActionKeyword) "(" [ActionText](#TActionText) {[StringOp](#TStringOp) [ActionText](#TActionText)}

 ")"

ActionKeyword ::= "action"

ActionText ::= [FreeText](#TFreeText) | [Expression](#TExpression)

ReturnStatement ::= [ReturnKeyword](#TReturnKeyword) [[TemplateInstance](#TTemplateInstance)]

/\* STATIC SEMANTICS - TemplateInstance shall evaluate to a value of a type compatible with the return type for functions returning a value. It shall evaluate to a value, template (literal or template instance), or a matching mechanism compatible with the return type for functions returning a template. \*/

AltConstruct ::= [AltKeyword](#TAltKeyword) "{" [AltGuardList](#TAltGuardList) "}"

AltKeyword ::= "alt"

AltGuardList ::= {[GuardStatement](#TGuardStatement) | [ElseStatement](#TElseStatement) [[SemiColon](#TSemiColon)]}

GuardStatement ::= [AltGuardChar](#TAltGuardChar) ([AltstepInstance](#TAltstepInstance) [[StatementBlock](#TStatementBlock)] |

 [GuardOp](#TGuardOp) [StatementBlock](#TStatementBlock))

ElseStatement ::= "[" [ElseKeyword](#TElseKeyword) "]" [StatementBlock](#TStatementBlock)

AltGuardChar ::= "[" [[BooleanExpression](#TBooleanExpression)] "]"

GuardOp ::= [TimeoutStatement](#TTimeoutStatement) |

 [ReceiveStatement](#TReceiveStatement) |

 [TriggerStatement](#TTriggerStatement) |

 [GetCallStatement](#TGetCallStatement) |

 [CatchStatement](#TCatchStatement) |

 [CheckStatement](#TCheckStatement) |

 [GetReplyStatement](#TGetReplyStatement) |

 [DoneStatement](#TDoneStatement) |

 [KilledStatement](#TKilledStatement)

InterleavedConstruct ::= [InterleavedKeyword](#TInterleavedKeyword) "{" [InterleavedGuardList](#TInterleavedGuardList)

 "}"

InterleavedKeyword ::= "interleave"

InterleavedGuardList ::= {[InterleavedGuardElement](#TInterleavedGuardElement) [[SemiColon](#TSemiColon)]}+

InterleavedGuardElement ::= [InterleavedGuard](#TInterleavedGuard) [StatementBlock](#TStatementBlock)

InterleavedGuard ::= "[" "]" [GuardOp](#TGuardOp)

LabelStatement ::= [LabelKeyword](#TLabelKeyword) [Identifier](#TIdentifier)

LabelKeyword ::= "label"

GotoStatement ::= [GotoKeyword](#TGotoKeyword) [Identifier](#TIdentifier)

GotoKeyword ::= "goto"

RepeatStatement ::= "repeat"

ActivateOp ::= [ActivateKeyword](#TActivateKeyword) "(" [AltstepInstance](#TAltstepInstance) ")"

ActivateKeyword ::= "activate"

DeactivateStatement ::= [DeactivateKeyword](#TDeactivateKeyword) ["(" [ObjectReference](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5C05PWRTCH%5CCR7707-v5.docx#TObjectReference) ")"]

DeactivateKeyword ::= "deactivate"

BreakStatement ::= "break"

ContinueStatement ::= "continue"

#### A.1.6.8.3 Basic statements

BasicStatements ::= [Assignment](#TAssignment) |

 [LogStatement](#TLogStatement) |

 [LoopConstruct](#TLoopConstruct) |

 [ConditionalConstruct](#TConditionalConstruct) |

 [SelectCaseConstruct](#TSelectCaseConstruct) |

 [StatementBlock](#TStatementBlock)

Expression ::= [SingleExpression](#TSingleExpression) | [CompoundExpression](#TCompoundExpression)

CompoundExpression ::= [FieldExpressionList](#TFieldExpressionList) | [ArrayOrMixedExpression](#TArrayExpression)

/\* STATIC SEMANTICS - Within CompoundExpression the ArrayOrMixedExpression can be used for Arrays, record, record of, set and set of types. \*/

FieldExpressionList ::= "{" [FieldExpressionSpec](#TFieldExpressionSpec) {"," [FieldExpressionSpec](#TFieldExpressionSpec)}

 "}"

FieldExpressionSpec ::= [FieldReference](#TFieldReference) [AssignmentChar](#TAssignmentChar) [NotUsedOrExpression](#TNotUsedOrExpression)

ArrayOrMixedExpression ::= "{" [[ArrayElementExpressionList](#TArrayElementExpressionList) {"," [FieldExpressionSpec](file:///C%3A%5CUsers%5Cethgry%5CAppData%5CLocal%5CMicrosoft%5CWindows%5CTemporary%20Internet%20Files%5CContent.IE5%5CSJHEOEMY%5CCR7655.docx#TFieldExpressionSpec)}] "}"

ArrayElementExpressionList ::= [NotUsedOrExpression](#TNotUsedOrExpression) {"," [NotUsedOrExpression](#TNotUsedOrExpression)}

NotUsedOrExpression ::= [Expression](#TExpression) | [Minus](#TMinus)

ConstantExpression ::= [SingleExpression](#TSingleExpression) | [CompoundConstExpression](#TCompoundConstExpression)

BooleanExpression ::= [SingleExpression](#TSingleExpression)

/\* STATIC SEMANTICS - BooleanExpression shall resolve to a Value of type Boolean \*/

CompoundConstExpression ::= [FieldConstExpressionList](#TFieldConstExpressionList) | [ArrayConstExpression](#TArrayConstExpression)

/\* STATIC SEMANTICS - Within CompoundConstExpression the ArrayConstExpression can be used for arrays, record, record of and set of types. \*/

FieldConstExpressionList ::= "{" [FieldConstExpressionSpec](#TFieldConstExpressionSpec) {"," [FieldConstExpressionSpec](#TFieldConstExpressionSpec)} "}"

FieldConstExpressionSpec ::= [FieldReference](#TFieldReference) [AssignmentChar](#TAssignmentChar) [ConstantExpression](#TConstantExpression)

ArrayConstExpression ::= "{" [[ArrayElementConstExpressionList](#TArrayElementConstExpressionList)] "}"

ArrayElementConstExpressionList ::= [ConstantExpression](#TConstantExpression) {"," [ConstantExpression](#TConstantExpression)}

Assignment ::= ValueRef [AssignmentChar](#TAssignmentChar) [TemplateBody](#TTemplateBody)

/\* STATIC SEMANTICS - The Templatebody on the right hand side of Assignment shall evaluate to an explicit value of a type compatible with the type of the left hand side for value variables and shall evaluate to an explicit value, template (literal or a template instance) or a matching mechanism compatible with the type of the left hand side for template variables. \*/

SingleExpression ::= [XorExpression](#TXorExpression) {"or" [XorExpression](#TXorExpression)}

/\* STATIC SEMANTICS - If more than one XorExpression exists, then the XorExpressions shall evaluate to specific values of compatible types \*/

XorExpression ::= [AndExpression](#TAndExpression) {"xor" [AndExpression](#TAndExpression)}

/\* STATIC SEMANTICS - If more than one AndExpression exists, then the AndExpressions shall evaluate to specific values of compatible types \*/

AndExpression ::= [NotExpression](#TNotExpression) {"and" [NotExpression](#TNotExpression)}

/\* STATIC SEMANTICS - If more than one NotExpression exists, then the NotExpressions shall evaluate to specific values of compatible types \*/

NotExpression ::= ["not"] [EqualExpression](#TEqualExpression)

/\* STATIC SEMANTICS - Operands of the not operator shall be of type boolean or derivatives of type Boolean. \*/

EqualExpression ::= [RelExpression](#TRelExpression) {[EqualOp](#TEqualOp) [RelExpression](#TRelExpression)}

/\* STATIC SEMANTICS - If more than one RelExpression exists, then the RelExpressions shall evaluate to specific values of compatible types. If only one RelExpression exists, it shall not derive to a CompoundExpression. \*/

RelExpression ::= [ShiftExpression](#TShiftExpression) [[RelOp](#TRelOp) [ShiftExpression](#TShiftExpression)] | [CompoundExpression](#TCompoundExpression)

/\* STATIC SEMANTICS - If both ShiftExpressions exist, then each ShiftExpression shall evaluate to a specific integer, Enumerated or float Value or derivatives of these types \*/

ShiftExpression ::= [BitOrExpression](#TBitOrExpression) {[ShiftOp](#TShiftOp) [BitOrExpression](#TBitOrExpression)}

/\* STATIC SEMANTICS - Each Result shall resolve to a specific Value. If more than one Result exists the right-hand operand shall be of type integer or derivatives and if the shift op is "<<" or ">>" then the left-hand operand shall resolve to either bitstring, hexstring or octetstring type or derivatives of these types. If the shift op is " \*/

BitOrExpression ::= [BitXorExpression](#TBitXorExpression) {"or4b" [BitXorExpression](#TBitXorExpression)}

/\* STATIC SEMANTICS - If more than one BitXorExpression exists, then the BitXorExpressions shall evaluate to specific values of compatible types \*/

BitXorExpression ::= [BitAndExpression](#TBitAndExpression) {"xor4b" [BitAndExpression](#TBitAndExpression)}

/\* STATIC SEMANTICS - If more than one BitAndExpression exists, then the BitAndExpressions shall evaluate to specific values of compatible types \*/

BitAndExpression ::= [BitNotExpression](#TBitNotExpression) {"and4b" [BitNotExpression](#TBitNotExpression)}

/\* STATIC SEMANTICS - If more than one BitNotExpression exists, then the BitNotExpressions shall evaluate to specific values of compatible types \*/

BitNotExpression ::= ["not4b"] [AddExpression](#TAddExpression)

/\* STATIC SEMANTICS - If the not4b operator exists, the operand shall be of type bitstring, octetstring or hexstring or derivatives of these types. \*/

AddExpression ::= [MulExpression](#TMulExpression) {[AddOp](#TAddOp) [MulExpression](#TMulExpression)}

/\* STATIC SEMANTICS - Each MulExpression shall resolve to a specific Value. If more than one MulExpression exists and the AddOp resolves to StringOp then the MulExpressions shall be valid operands for StringOp. If more than one MulExpression exists and the AddOp does not resolve to StringOp then the MulExpression shall both resolve to type integer or float or derivatives of these types. If only one MulExpression exists, it shall not derive to a CompoundExpression. \*/

MulExpression ::= [UnaryExpression](#TUnaryExpression) {[MultiplyOp](#TMultiplyOp) [UnaryExpression](#TUnaryExpression)} | [CompoundExpression](#TCompoundExpression)

/\* STATIC SEMANTICS - Each UnaryExpression shall resolve to a specific Value. If more than one UnaryExpression exists then the UnaryExpressions shall resolve to type integer or float or derivatives of these types. \*/

UnaryExpression ::= [[UnaryOp](#TUnaryOp)] [Primary](#TPrimary)

/\* STATIC SEMANTICS - Primary shall resolve to a specific Value of type integer or float or derivatives of these types.\*/

Primary ::= [OpCall](#TOpCall) |

 [Value](#TValue) |

 "(" [SingleExpression](#TSingleExpression) ")"

ExtendedFieldReference ::= {([Dot](#TDot) ([Identifier](#TIdentifier) | [PredefinedType](#TPredefinedType))) |

 [ArrayOrBitRef](#TArrayOrBitRef) |

 [DecodedFieldReference](#TDecodedFieldReference)

 }+

/\* STATIC SEMANTIC - The Identifier refers to a type definition if the type of the VarInstance or ReferencedValue in which the ExtendedFieldReference is used is anytype. ArrayOrBitRef shall be used when referencing elements of values or arrays. DecodedFieldReference shall not appear on the LHS of assignments and in type references\*/

 DecodedFieldReference ::= "=>" [DecodedFieldType](#TDecodedFieldType)

 DecodedFieldType ::= [PredefinedType](#TPredefinedType) |

 [Identifier](#TIdentifier) |

 "(" [Type](#TType) [ "," [Expression](#TExpression) ] ")"

/\* The Identifier shall resolve into a type \*/

OpCall ::= [ConfigurationOps](#TConfigurationOps) |

 [GetLocalVerdict](#TGetLocalVerdict) |

 [TimerOps](#TTimerOps) |

 [TestcaseInstance](#TTestcaseInstance) |

 ([FunctionInstance](#TFunctionInstance) [[ExtendedFieldReference](#TExtendedFieldReference)]) |

 ([TemplateOps](#TTemplateOps) [[ExtendedFieldReference](#TExtendedFieldReference)]) |

 [ActivateOp](#TActivateOp) |

 [GetAttributeOp](#TGetAttributeOp)

AddOp ::= "+" |

 "-" |

 [StringOp](#TStringOp)

/\* STATIC SEMANTICS - Operands of the "+" or "-" operators shall be of type integer or float or derivations of integer or float (i.e. subrange) \*/

MultiplyOp ::= "\*" | "/" | "mod" | "rem"

/\* STATIC SEMANTICS - Operands of the "\*", "/", rem or mod operators shall be of type integer or float or derivations of integer or float (i.e. subrange) \*/

UnaryOp ::= "+" | "-"

/\* STATIC SEMANTICS - Operands of the "+" or "-" operators shall be of type integer or float or derivations of integer or float (i.e. subrange) \*/

RelOp ::= "<" | ">" | ">=" | "<="

/\* STATIC SEMANTICS - the precedence of the operators is defined in Table 6 \*/

EqualOp ::= "==" | "!="

StringOp ::= "&"

/\* STATIC SEMANTICS - Operands of the list operator shall be bitstring, hexstring, octetstring, (universal) character string, record of, set of, or array types, or derivates of these types \*/

ShiftOp ::= "<<" | ">>" | "<@" | "@>"

LogStatement ::= [LogKeyword](#TLogKeyword) "(" [LogItem](#TLogItem) {"," [LogItem](#TLogItem)} ")"

LogKeyword ::= "log"

LogItem ::= [FreeText](#TFreeText) | [TemplateInstance](#TTemplateInstance)

LoopConstruct ::= [ForStatement](#TForStatement) |

 [WhileStatement](#TWhileStatement) |

 [DoWhileStatement](#TDoWhileStatement)

ForStatement ::= [ForKeyword](#TForKeyword) "(" [Initial](#TInitial) [SemiColon](#TSemiColon) [BooleanExpression](#TBooleanExpression)

 [SemiColon](#TSemiColon) [Assignment](#TAssignment) ")" [StatementBlock](#TStatementBlock)

ForKeyword ::= "for"

Initial ::= [VarInstance](#TVarInstance) | [Assignment](#TAssignment)

WhileStatement ::= [WhileKeyword](#TWhileKeyword) "(" [BooleanExpression](#TBooleanExpression) ")" [StatementBlock](#TStatementBlock)

WhileKeyword ::= "while"

DoWhileStatement ::= [DoKeyword](#TDoKeyword) [StatementBlock](#TStatementBlock) [WhileKeyword](#TWhileKeyword) "(" [BooleanExpression](#TBooleanExpression)

 ")"

DoKeyword ::= "do"

ConditionalConstruct ::= [IfKeyword](#TIfKeyword) "(" [BooleanExpression](#TBooleanExpression) ")" [StatementBlock](#TStatementBlock)

 {[ElseIfClause](#TElseIfClause)} [[ElseClause](#TElseClause)]

IfKeyword ::= "if"

ElseIfClause ::= [ElseKeyword](#TElseKeyword) [IfKeyword](#TIfKeyword) "(" [BooleanExpression](#TBooleanExpression) ")" [StatementBlock](#TStatementBlock)

ElseKeyword ::= "else"

ElseClause ::= [ElseKeyword](#TElseKeyword) [StatementBlock](#TStatementBlock)

SelectCaseConstruct ::= [SelectKeyword](#TSelectKeyword) [[UnionKeyword](#TUnionKeyword)] "(" [SingleExpression](#TSingleExpression) ")" [SelectCaseBody](#TSelectCaseBody)

SelectKeyword ::= "select"

SelectCaseBody ::= "{" {[SelectCase](#TSelectCase)}+ [[CaseElse](#TCaseElse)] "}"

SelectCase ::= [CaseKeyword](#TCaseKeyword) ("("[TemplateInstance](#TTemplateInstance) {"," [TemplateInstance](#TTemplateInstance)}

 ")" | [ElseKeyword](#TElseKeyword)) [StatementBlock](#TStatementBlock)

/\*\* STATIC SEMANTICS TemplateInstance-s shall be Identifier-s if the UnionKeyword is present in the surrounding SelectCaseConstruct (see clause 19.3.2)\*/

CaseElse ::= [CaseKeyword](#TCaseKeyword) [ElseKeyword](#TElseKeyword) [StatementBlock](#TStatementBlock)

CaseKeyword ::= "case"

ExtendedIdentifier ::= [[Identifier](#TIdentifier) [Dot](#TDot)] [Identifier](#TIdentifier)

IdentifierList ::= [Identifier](#TIdentifier) {"," [Identifier](#TIdentifier)}

QualifiedIdentifierList ::= [QualifiedIdentifier](#TQualifiedIdentifier) {"," [QualifiedIdentifier](#TQualifiedIdentifier)}

GetAttributeOp ::= ([Type](#TType) | [TemplateInstance](#TTemplateInstance)) "." [GetAttributeSpec](#TGetAttributeSpec)

GetAttributeSpec ::= [EncodeKeyword](#TEncodeKeyword) |

 [VariantKeyword](#TVariantKeyword) ["(" [FreeText](#TFreeText) ")"] |

 [DisplayKeyword](#TDisplayKeyword) |

 [ExtensionKeyword](#TExtensionKeyword) |

 [OptionalKeyword](#TOptionalKeyword)

### A.1.6.9 Miscellaneous productions

Dot ::= "."

Minus ::= "-"

SemiColon ::= ";"

Colon ::= ":"

Underscore ::= "\_"

AssignmentChar ::= ":="

IndexModifier ::= "@index"

DeterministicModifier ::= "@deterministic"

LazyModifier ::= "@lazy"

FuzzyModifier ::= "@fuzzy"

CaseInsenModifier ::= "@nocase"

DecodedModifier ::= "@decoded"

DefaultModifier ::= "@default"

ControlModifier ::= "@control"