Final draft ETSI ES 201 873-1 V4.8.1 (2016-05)

Methods for Testing and Specification (MTS);

The Testing and Test Control Notation version 3;

Part 1: TTCN‑3 Core Language

**ETSI Standard**

Reference

RES/MTS-201873-1 T3ed481

Keywords

language, methodology, testing, TTCN-3

***ETSI***

650 Route des Lucioles

F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C

Association à but non lucratif enregistrée à la

Sous-Préfecture de Grasse (06) N° 7803/88

***Important notice***

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

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

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

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

***Copyright Notification***

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

© European Telecommunications Standards Institute 2016.

All rights reserved.

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

### 5.4.2 Actual parameters

Values, templates, timers and/or ports can be passed into parameterized TTCN-3 objects as actual parameters. Actual parameters can be provided both as a list in the same order as the formal parameters as well as in an assignment notation explicitly using the associated formal parameter names or in a mixed notation where the first parameters are given in list notation and additional parameters in assignment notation.

***Syntactical Structure***

( *Expression* | // for value parameter

 *TemplateInstance* | // for template parameter

 *TimerRef* | // for timer parameter

 *Port* | // for port parameter

 "-" ) | // to skip a parameter with default
 *ParameterId* ":=" ( Expression | TemplateInstance | TimerRef | Port ) )

***Semantic Description***

Actual parameters that are passed by value to **in** formal value parameters shall be variables, literal values, module parameters, constants, value variables, invocations of value returning (external) functions, formal value parameters (of in, inout or out parameterization) of the current scope or expressions composed of the above.

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

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

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

Actual parameters that are passed to **in** formal template parameters shall be literal values, module parameters, constants, variables, invocations of value or template returning (external) functions, formal value parameters (of in, inout or out parameterization) of the current scope or expressions composed of the above, as well as templates, template variables or formal template parameters (of in, inout or out parameterization) of the current scope.

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

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

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

The values of **out** formal parameters are passed to the actual parameters in the same order as is the order of formal parameters in the definition of the parameterized TTCN-3 object. The value is passed back to the actual parameter only if within the invoked object a value is assigned to it. If no value is assigned, the actual parameter remains unchanged when the invoked object completes.

Actual parameters that are passed to formal timer parameters shall be component timers, local timers or formal timer parameters of the current scope.

Actual parameters that are passed to formal port parameters shall be component ports or formal port parameters of the current scope.

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

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

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

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

NOTE 3: If assignment notation has been used for the actual parameter list, the order of the evaluation of actual parameters may differ from the order of the parameters in the formal parameter list.

The empty brackets for instances of parameterized templates that have only parameters with default values are optional when no actual parameters are provided, i.e. all formal parameters use their default values.

***Restrictions***

a) When using list notation, the order of elements in the actual parameter list shall be the same as their order in the corresponding formal parameter list. For each formal **inout** parameter and for each **in** parameter without a default there shall be an actual parameter. The actual parameter of a formal **out** parameter or **in** parameter with default value can be skipped by using dash "-" as actual parameter. An actual parameter can also be skipped by just leaving it out if no other actual parameter follows in the actual parameter list - either because the parameter is last or because all following formal parameters are **out** parameters or have default values and are left out. The number of actual parameters in the list notation shall not exceed the number of parameters in the formal parameter list.

b) Either list notation or assignment notation shall be used in a single parameter list. They shall not be mixed.

c) When using assignment notation, each formal parameter shall be assigned an actual parameter at most once. For each assigned actual parameter there shall exist a corresponding formal parameter of the same name. For each formal parameter without default value, there shall be an actual parameter. In order to use the default value of a formal parameter, no assignment for this specific parameter shall be provided.

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

e) Actual parameters passed to restricted formal template parameters shall obey the restrictions given in clause 15.8.

f) All parameterized entities specified as an actual parameter shall have their own parameters resolved in the top‑level actual parameter list.

g) If the formal parameter list of TTCN‑3 objects **function**, **testcase**, **altstep** or **external** **function** is empty, then the empty parentheses shall be included both in the declaration and in the invocation of that object. In all other cases the empty parentheses shall be omitted.

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

h) Void.

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

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

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

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

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

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

m) If the mixed notation is used, the parameters given in assignment notation shall not assign parameters that already have an actual parameter given in list notation.

***Examples***

EXAMPLE 1: Formal and actual parameter lists have to match

 // A function definition with a formal parameter list

 **function** f\_myFunction(**integer** p\_formalPar1, **boolean** p\_formalPar2, **bitstring** p\_formalPar3) { … }

 // A function call with an actual parameter list

 f\_myFunction(123, **true**,'1100'B);

 // A function call with assignment notation for actual parameters

 f\_myFunction(p\_formalPar1 := 123, p\_formalPar3 := '1100'B, p\_formalPar2 := **true**);

EXAMPLE 2: In parameters

 **function** f\_myFunction(**in** **template** MyTemplateType p\_myValueParameter){ … };

 // p\_myValueParameter is in parameter, the in keyword is optional

 // A function call with an actual parameter

 f\_myFunction(m\_myGlobalTemplate);

EXAMPLE 3: Inout and out parameters

 **function** f\_myFunction(**inout boolean** p\_myReferenceParameter){ … };

 // p\_myReferenceParameter is an inout parameter

 // A function call with an actual parameter

 f\_myFunction(v\_myBooleanVariable);

 // The actual parameter can be read and set within the function

 **function** f\_myFunction(**out template boolean** p\_myReferenceParameter){ … };

 // p\_myReferenceParameter is an out parameter

 // A function call with an actual parameter

 f\_myFunction(v\_myBooleanVariable);

 // The actual parameter is initially unbound, but can be set and read within the function.

 f\_myFunction(**-**); // the outcoming value is not assigned to a variable

 **type** **record** **of** **integer** RoI;

 **function** f\_swapElements (**inout integer** p\_int1, **inout** **integer** p\_int2) {

 **var** **integer** v\_tmp := p\_int1;

 p\_int1 := p\_int2;

 p\_int2 := v\_tmp;

 }

 **function** f\_testReferences (**inout** RoI p\_roi, **inout** **integer** p\_elem) { … }

 :

 **var** RoI v\_roi := { 0, 1, 2, 3, 4, 5 };

 f\_swapElements(v\_roi[0], v\_roi[5]); // after the function call, v\_roi is { 5, 1, 2, 3, 4, 0 }

 f\_testReferences(v\_roi, v\_roi[2]); // produces an error as elements of v\_roi are not allowed

 // to be passed by reference if the parent structure (v\_roi) is passed by reference too.

 **function** f\_changeAndIncrement(**out** **integer** p\_e, **in** **integer** p\_v, **inout** **integer** p\_i) {

 p\_i := p\_i + 1;

 p\_e := p\_v;

 }

 :

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

 f\_changeAndIncrement(v\_roi[v\_i], 3, v\_i); // increments p\_i, but still assigns 3 to v\_roi[0]

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

 **type** **component** MyComp {

 **var** ROI v\_rec := { 0, 1 }

 }

 **testcase** TC() **runs on** MyComp {

 f\_test(v\_rec[1]) // passing 2nd element of component variable as inout parameter

 **log**(v\_rec); //will log { 2 , 2 }

 }

 **function** f\_test(**inout** **integer** p\_int) **runs on** MyComp {

 v\_rec := { 2 }; // now, isbound(p\_int) == false

 p\_int := 2; // now, v\_rec == { 2, 2 }

 }

EXAMPLE 5: Empty parameter lists

 // A function definition with an empty parameter list shall be written as

 **function** f\_myFunction(){ … }

 // and shall be called as

f\_myFunction();

 // A template definition with a default value for a formal parameter written as

 **template** MyRecord m\_mytemplate (**integer** p\_myValue:= 1):= { … }

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

myPCO**.send**(m\_mytemplate)

EXAMPLE 6: Nested parameter lists

 // Given the message definition

 **type record** MyMessageType

 {

 **integer** field1,

 **charstring** field2,

 **boolean** field3

 }

 // A message template might be

 **template** MyMessageType mw\_myTemplate(**integer** p\_myValue) :=

 {

 field1 := p\_myValue,

 field2 := **pattern** "abc\*xyz",

 field3 := **true**

 }

 // A test case parameterized with a template might be

**testcase** TC\_001(**template** MyMessageType p\_rxMsg) **runs on** PTC1 **system** TS1 {

 :

 myPCO.**receive**(p\_rxMsg);

 }

 // When the test case is called in the control part and the parameterized template is

 // passed as an actual parameter, the template's actual parameters shall be provided

 **control**

{ :

 **execute**(TC\_001(mw\_myTemplate(7)));

 :

 }

EXAMPLE 7: A typical use case for lazy parameterization

 **modulepar** **boolean** PX\_LOG\_MESSAGE := **true**;

 **function** f\_logMsg(**@lazy** **charstring** p\_complex) {

 **if** (PX\_LOG\_MESSAGE) {

 **log**(p\_complex);

 }

 }

 **function** f\_computeComplexMessage() **return charstring** {

 // some complicated computation

 }

 f\_logMsg(f\_computeComplexMessage()); // f\_computeComplexMessage() is only invoked if

 // PX\_LOG\_MESSAGE is true

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

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

 **type** **port** MyPortType **message** { **inout** MyMessage }

 **type** **component** MyMTC {

 **var** **integer** vc\_id;

 **port** MyPortType p;

 }

 **testcase** TC\_shootingMessages () **runs** **on** MyMTC {

 **connect**(**self**:p,**self**:p);

 f\_sendLazy({vc\_id, **rnd**()}); //note that at this point vc\_id is unintialized yet

 f\_sendFuzzy({vc\_id, **rnd**()})

 }

 **function** f\_sendLazy(**@lazy** MyMessage p\_pdu) **runs** **on** MyMTC {

 **for** (vc\_id := 1; vc\_id<9; vc\_id:=vc\_id+1){

 p.**send**(p\_pdu); // the actual parameter passed to the formal parameter p\_pdu is evaluated only

 // in the first loop;let say rnd() returns 0.924946;

 // the message { 1, 0.924946 } is sent out 8 times

 }

 **setverdict**(**pass**,"messages has been sent out")

 }

 **function** f\_sendFuzzy(**@fuzzy** MyMessage p\_pdu) **runs** **on** MyMTC {

 **for** (vc\_id := 1; vc\_id<9; vc\_id:=vc\_id+1){

 p.**send**(pdu); // the actual parameter passed to the formal parameter p\_pdu is evaluated in each

 // loop; let say rnd() returns 0.924946, 0.680497, 0.630836, 0.648681, 0.428501,

 // 0.262539, 0.646990, 0.265262 in subsuent calls; the messages 1, 0.924946 },

 // {{ 2, 0.680497 }, { 3, 0.630836 }, { 4, 0.648681 }, { 5, 0.428501 },

 // { 6, 0.262539 }, { 7, 0.646990 } and { 8, 0.265262 } are sent out in sequence

 }

 **setverdict**(**pass**,"messages has been sent out")

 }

EXAMPLE 9: Order of out parameters

 **function** f\_initValues (**out** **integer** p\_par1, **out** **integer** p\_par2) {

 p\_par1 := 1**;**

 p\_par2 := 2;

 }

 **function** f\_f(){

 **var** **integer** v\_var1;

 f\_initValues(p\_par2 := v\_var1, p\_par1 := v\_var1);

 // After this function call, v\_var1 will contain 2, as parameters are assigned in

 // the same order as in the definition of the f\_initValues function. Thus p\_par1 is

 // assigned first to v\_var1 and p\_par2 after that overwriting the previous value.

 }

EXAMPLE 10: Skipped actual parameters

 **function** f\_skip (**out** **integer** p\_par1, **in** **integer** p\_par2 := 2) {

 p\_par1 := 1 + p\_par2**;**

 }

 **function** f\_f(){

 // the following statements all have the same semantics :

 f\_skip (-,-); // p\_par2 is initialized with default value 2 and

 // the result of p\_par1 is not assigned to any variable

 f\_skip (p\_par1 := -, p\_par2 := -);

 f\_skip (p\_par2 := -); // skip p\_par1

 f\_skip (-) ; // skip p\_par2 because it is the last

 f\_skip () ; // skip p\_par1 because all following are also skipped

 }

EXAMPLE 11: Mixed notation

 **function** f\_mixed (**out** **integer** p\_par1, **in** **integer** p\_par2 := 2, inout integer p\_par3) {

 p\_par1 := 1 + p\_par2**;**

 }

 **function** f\_f(){

 var integer v := 0;

 // the following statements all have the same semantics :

 f\_mixed(-,2,v);

 f\_mixed(-,p\_par2 := 2, p\_par3 := v);

 f\_mixed(-,-,p\_par3 := v);

 f\_mixed(-,p\_par3 := v, p\_par2 := 2);

 // not allowed:

 f\_mixed(-,2,p\_par3 := v, p\_par2 := 5); // p\_par2 is already assigned in list notation

 }

## A.1.6 TTCN-3 syntax BNF productions

### A.1.6.0 TTCN-3 module

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

 [[ModuleControlPart](#TModuleControlPart)] "}" [[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) |

 [ExtConstDef](#TExtConstDef)

 )) |

 (["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 ::= ([Identifier](#TIdentifier) | [AddressKeyword](#TAddressKeyword)) "{" [[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 ::= ([Identifier](#TIdentifier) | [AddressKeyword](#TAddressKeyword)) "{" [UnionFieldDef](#TUnionFieldDef) {","

 [UnionFieldDef](#TUnionFieldDef)}

 "}"

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

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

SetKeyword ::= "set"

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

OfKeyword ::= "of"

StructOfDefBody ::= ([Type](#TType) | [NestedTypeDef](#TNestedTypeDef)) ([Identifier](#TIdentifier) | [AddressKeyword](#TAddressKeyword))

 [[SubTypeSpec](#TSubTypeSpec)]

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

EnumDef ::= [EnumKeyword](#TEnumKeyword) ([Identifier](#TIdentifier) | [AddressKeyword](#TAddressKeyword)) "{" [EnumerationList](#TEnumerationList)

 "}"

EnumKeyword ::= "enumerated"

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

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

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

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

SubTypeDef ::= [Type](#TType) ([Identifier](#TIdentifier) | [AddressKeyword](#TAddressKeyword)) [[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)]

 [BaseTemplate](#TBaseTemplate) [[DerivedDef](#TDerivedDef)] [AssignmentChar](#TAssignmentChar) [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 ::= ([SimpleSpec](#TSimpleSpec) |

 [FieldSpecList](#TFieldSpecList) |

 [ArrayValueOrAttrib](#TArrayValueOrAttrib)

 ) [[ExtraMatchingAttributes](#TExtraMatchingAttributes)]

/\* STATIC SEMANTICS - Within TeplateBody 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 ::= "(" [TemplateBody](#TTemplateBody) {"," [TemplateBody](#TTemplateBody) } ")"

/\*\* STATIC Semantics: each TemplateBody shall be an integer template \*/

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

DecodedContentMatch ::= DecodedMatchKeyword ["(" Expression] ")"] TemplateInstance
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] [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"

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

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

TemplateRefWithParList ::= [ExtendedIdentifier](#TExtendedIdentifier) [[TemplateActualParList](#TTemplateActualParList)]

TemplateInstance ::= [([Type](#TType) | [Signature](#TSignature)) [Colon](#TColon)] [[DerivedRefWithParList](#TDerivedRefWithParList) [AssignmentChar](#TAssignmentChar)]

 [TemplateBody](#TTemplateBody)

DerivedRefWithParList ::= [ModifiesKeyword](#TModifiesKeyword) [TemplateRefWithParList](#TTemplateRefWithParList)

TemplateActualParList ::= "(" [([TemplateInstanceActualPar](#TTemplateInstanceActualPar) {"," [TemplateInstanceActualPar](#TTemplateInstanceActualPar)}

 {"," [TemplateInstanceAssignment](#TTemplateInstanceAssignment)})

 | ([TemplateInstanceAssignment](#TTemplateInstanceAssignment) {"," [TemplateInstanceAssignment](#TTemplateInstanceAssignment)})]

 ")"

TemplateInstanceActualPar ::= [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 ")"

ValueofKeyword ::= "valueof"

#### A.1.6.1.4 Function definitions

FunctionDef ::= [FunctionKeyword](#TFunctionKeyword) [[DeterministicModifier](#TDeterministicModifier)] [Identifier](#TIdentifier)

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

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

FunctionKeyword ::= "function"

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

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

 [FormalTimerPar](#TFormalTimerPar) |

 [FormalTemplatePar](#TFormalTemplatePar) |

 [FormalPortPar](#TFormalPortPar)

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

 [Type](#TType)

ReturnKeyword ::= "return"

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

RunsKeyword ::= "runs"

OnKeyword ::= "on"

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

MTCKeyword ::= "mtc"

StatementBlock ::= "{" [[FunctionDefList](#TFunctionDefList)] [[FunctionStatementList](#TFunctionStatementList)] "}"

FunctionDefList ::= {([FunctionLocalDef](#TFunctionLocalDef) | [FunctionLocalInst](#TFunctionLocalInst)) [[WithStatement](#TWithStatement)]

 [[SemiColon](#TSemiColon)]}+

FunctionStatementList ::= {[FunctionStatement](#TFunctionStatement) [[SemiColon](#TSemiColon)]}+

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) "(" [[FunctionActualParList](#TFunctionActualParList)] ")"

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

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

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

FunctionActualParList ::= ([FunctionActualPar](#TFunctionActualPar) {"," [FunctionActualPar](#TFunctionActualPar)}

 {"," [FunctionActualParAssignment](#TFunctionActualParAssignment)}) |

 ([FunctionActualParAssignment](#TFunctionActualParAssignment) {"," [FunctionActualParAssignment](#TFunctionActualParAssignment)})

FunctionActualPar ::= [ArrayIdentifierRef](#TArrayIdentifierRef) |

 [TemplateInstance](#TInLineTemplate) |

 [ComponentRef](#TComponentRef) |

 [Minus](#TMinus)

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

FunctionActualParAssignment ::= [TemplateInstanceAssignment](#TTemplateInstanceAssignment) |

 [ComponentRefAssignment](#TComponentRefAssignment) |

 [ArrayIdentifierRefAssignment](#TArrayIdentifierRefAssignment)

ArrayIdentifierRefAssignment ::= [Identifier](#TIdentifier) ":=" [ArrayIdentifierRef](#TArrayIdentifierRef)

#### 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) "(" [[TestcaseActualParList](#TTestcaseActualParList)]

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

 ")"

ExecuteKeyword ::= "execute"

TestcaseActualParList ::= ([TemplateInstanceActualPar](#TTemplateInstanceActualPar) {"," [TemplateInstanceActualPar](#TTemplateInstanceActualPar)}

 {"," [TemplateInstanceAssignment](#TTemplateInstanceAssignment)}) |

 ([TemplateInstanceAssignment](#TTemplateInstanceAssignment) {"," [TemplateInstanceAssignment](#TTemplateInstanceAssignment)})

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

#### A.1.6.1.7 Altstep definitions

AltstepDef ::= [AltstepKeyword](#TAltstepKeyword) [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) "(" [[FunctionActualParList](#TFunctionActualParList)]

 ")"

#### A.1.6.1.8 Import definitions

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

ExceptTypeDefSpec ::= [TypeDefKeyword](#TTypeDefKeyword) [IdentifierListOrAll](#TIdentifierListOrAll)

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

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

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

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

ExceptFunctionSpec ::= [FunctionKeyword](#TFunctionKeyword) [IdentifierListOrAll](#TIdentifierListOrAll)

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) [[RecursiveKeyword](#TRecursiveKeyword)]

RecursiveKeyword ::= "recursive"

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)

ImportTypeDefSpec ::= [TypeDefKeyword](#TTypeDefKeyword) [IdentifierListOrAllWithExcept](#TIdentifierListOrAllWithExcept)

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

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

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

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

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

ImportFunctionSpec ::= [FunctionKeyword](#TFunctionKeyword) [IdentifierListOrAllWithExcept](#TIdentifierListOrAllWithExcept)

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

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

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

#### 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 External constant definitions

ExtConstDef ::= [ExtKeyword](#TExtKeyword) [ConstKeyword](#TConstKeyword) [Type](#TType) [IdentifierList](#TIdentifierList)

#### A.1.6.1.12 Module parameter definitions

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

 "}"))

ModuleParKeyword ::= "modulepar"

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

ModulePar ::= [Type](#TType) [ModuleParList](#TModuleParList)

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

 [Identifier](#TIdentifier)

 [[AssignmentChar](#TAssignmentChar)

 [ConstantExpression](#TConstantExpression)]}

#### A.1.6.1.13 Friend module definitions

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

### A.1.6.2 Control part

ModuleControlPart ::= [ControlKeyword](#TControlKeyword) "{" [ModuleControlBody](#TModuleControlBody) "}" [[WithStatement](#TWithStatement)]

 [[SemiColon](#TSemiColon)]

ControlKeyword ::= "control"

ModuleControlBody ::= [[ControlStatementOrDefList](#TControlStatementOrDefList)]

ControlStatementOrDefList ::= {[ControlStatementOrDef](#TControlStatementOrDef) [[SemiColon](#TSemiColon)]}+

ControlStatementOrDef ::= ([FunctionLocalDef](#TFunctionLocalDef) | [FunctionLocalInst](#TFunctionLocalInst)) [[WithStatement](#TWithStatement)] |

 [ControlStatement](#TControlStatement)

ControlStatement ::= [TimerStatements](#TTimerStatements) |

 [BasicStatements](#TBasicStatements) |

 [BehaviourStatements](#TBehaviourStatements) |

 [SUTStatements](#TSUTStatements) |

 [StopKeyword](#TStopKeyword)

### A.1.6.3 Local definitions

#### A.1.6.3.1 Variable instantiation

VarInstance ::= [VarKeyword](#TVarKeyword) (([[LazyModifier](#TLazyModifier) | [FuzzyModifier](#TFuzzyModifier)] [Type](#TType) [VarList](#TVarList)) |

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

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

VariableRef ::= [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) |

 [DoneStatement](#TDoneStatement) |

 [KilledStatement](#TKilledStatement) |

 [StartTCStatement](#TStartTCStatement) |

 [StopTCStatement](#TStopTCStatement) |

 [KillTCStatement](#TKillTCStatement)

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 ::= [ComponentOrDefaultReference](#TComponentOrDefaultReference) |

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

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

ValueStoreSpec ::= ValueKeyword VariableRef

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 ::= [ComponentOrDefaultReference](#TComponentOrDefaultReference) |

 [SystemKeyword](#TSystemKeyword) |

 [SelfOp](#TSelfOp) |

 [MTCKeyword](#TMTCKeyword)

ComponentRefAssignment ::= [Identifier](#TIdentifier) ":=" [ComponentRef](#TComponentRef)

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) [FunctionActualParList](#TFunctionActualParList)

MapKeyword ::= "map"

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

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

 [AllPortsSpec](#TAllPortsSpec) |

 [AllCompsAllPortsSpec](#TAllCompsAllPortsSpec)

 ]

UnmapKeyword ::= "unmap"

StartTCStatement ::= [ComponentOrDefaultReference](#TComponentOrDefaultReference) [Dot](#TDot) [StartKeyword](#TStartKeyword)

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

StartKeyword ::= "start"

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

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

ComponentReferenceOrLiteral ::= [ComponentOrDefaultReference](#TComponentOrDefaultReference) |

 [MTCKeyword](#TMTCKeyword) |

 [SelfOp](#TSelfOp)

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

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

 [KillKeyword](#TKillKeyword))

ComponentOrDefaultReference ::= [VariableRef](#TVariableRef) | [FunctionInstance](#TFunctionInstance)

KillKeyword ::= "kill"

#### A.1.6.4.2 Port operations

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

 [CallStatement](#TCallStatement) |

 [ReplyStatement](#TReplyStatement) |

 [RaiseStatement](#TRaiseStatement) |

 [ReceiveStatement](#TReceiveStatement) |

 [TriggerStatement](#TTriggerStatement) |

 [GetCallStatement](#TGetCallStatement) |

 [GetReplyStatement](#TGetReplyStatement) |

 [CatchStatement](#TCatchStatement) |

 [CheckStatement](#TCheckStatement) |

 [ClearStatement](#TClearStatement) |

 [StartStatement](#TStartStatement) |

 [StopStatement](#TStopStatement) |

 [HaltStatement](#THaltStatement) |

 [CheckStateStatement](#TCheckStateStatement)

SendStatement ::= [ArrayIdentifierRef](#TArrayIdentifierRef) [Dot](#TDot) [PortSendOp](#TPortSendOp)

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

SendOpKeyword ::= "send"

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

 [AddressRefList](#TAddressRefList) |

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

 )

AddressRefList ::= "(" TemplateInstance {"," TemplateInstance} ")"

ToKeyword ::= "to"

CallStatement ::= [ArrayIdentifierRef](#TArrayIdentifierRef) [Dot](#TDot) [PortCallOp](#TPortCallOp) [[PortCallBody](#TPortCallBody)]

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

CallOpKeyword ::= "call"

CallParameters ::= TemplateInstance ["," [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 ::= [ArrayIdentifierRef](#TArrayIdentifierRef) [Dot](#TDot) [PortReplyOp](#TPortReplyOp)

PortReplyOp ::= [ReplyKeyword](#TReplyKeyword) "(" TemplateInstance [[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 ::= [ArrayIdentifierRef](#TArrayIdentifierRef) [Dot](#TDot) [PortRaiseOp](#TPortRaiseOp)

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

 [[ToClause](#TToClause)]

RaiseKeyword ::= "raise"

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

PortOrAny ::= [ArrayIdentifierRef](#TArrayIdentifierRef) | ([AnyKeyword](#TAnyKeyword) ([PortKeyword](#TPortKeyword) | [FromKeyword](#TFromKeyword)

 [VariableRef](#TVariableRef)))

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

 [[PortRedirect](#TPortRedirect)]

ReceiveOpKeyword ::= "receive"

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

 [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) ([VariableRef](#TVariableRef) | ("(" [SingleValueSpec](#TSingleValueSpec) {"," [SingleValueSpec](#TSingleValueSpec)} ")"))

SingleValueSpec ::= [VariableRef](#TVariableRef) [[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) [VariableRef](#TVariableRef)

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 ")"] [[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 ::= [VariableRef](#TVariableRef) [AssignmentChar](#TAssignmentChar) [ DecodedModifier ["(" Expression] ")"]
 [Identifier](#TIdentifier)

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

VariableEntry ::= [VariableRef](#TVariableRef) | [Minus](#TMinus)

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

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

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 | [TimeoutKeyword](#TTimeoutKeyword)

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

PortOrAll ::= [ArrayIdentifierRef](#TArrayIdentifierRef) | [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) |

 [TimeoutStatement](#TTimeoutStatement)

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

StartTimerStatement ::= [ArrayIdentifierRef](#TArrayIdentifierRef) [Dot](#TDot) [StartKeyword](#TStartKeyword) ["(" [Expression](#TExpression)

 ")"]

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

TimerRefOrAll ::= [ArrayIdentifierRef](#TArrayIdentifierRef) | [AllKeyword](#TAllKeyword) [TimerKeyword](#TTimerKeyword)

ReadTimerOp ::= [ArrayIdentifierRef](#TArrayIdentifierRef) [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 ::= [ArrayIdentifierRef](#TArrayIdentifierRef) |

 ([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 [","] } ")"]

### 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)

BitStringKeyword ::= "bitstring"

BooleanKeyword ::= "boolean"

IntegerKeyword ::= "integer"

OctetStringKeyword ::= "octetstring"

HexStringKeyword ::= "hexstring"

VerdictTypeKeyword ::= "verdicttype"

FloatKeyword ::= "float"

AddressKeyword ::= "address"

DefaultKeyword ::= "default"

AnyTypeKeyword ::= "anytype"

CharStringKeyword ::= "charstring"

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

UniversalKeyword ::= "universal"

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

TypeReference ::= [ExtendedIdentifier](#TExtendedIdentifier)

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

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

### A.1.6.6 Value

Value ::= [PredefinedValue](#TPredefinedValue) | [ReferencedValue](#TReferencedValue)

PredefinedValue ::= [Bstring](#TBstring) |

 [BooleanValue](#TBooleanValue) |

 [CharStringValue](#TCharStringValue) |

 [Number](#TNumber) | /\* IntegerValue \*/

 [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

Quadruple ::= [CharKeyword](#TCharKeyword) "(" [Number](#TNumber) "," [Number](#TNumber) "," [Number](#TNumber) "," [Number](#TNumber) ")"

USIlikeNotation ::= [CharKeyword](#TCharKeyword) "(" UIDlike { "," [UID](#TNumber)like } ")"

UIDlike ::= (U|u) {"+"} {Hex}#(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) | [ExtendedEnumReference](#TExtendedEnumReference)]

/\*\* STATIC Semantics: ExtendedEnumReference shall be present if and only if ExtendedIdentifier refers to an enumerated value with an attached value list \*/

ExtendedEnumReference ::= "(" [IntegerValue](#TEnumValue) ")"

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 } "'" "B"

Bin ::= "0" | "1"

Hstring ::= "'" { [Hex](#THex) | BinSpace } "'" "H"

Hex ::= [Num](#TNum) | "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" |

 "d" | "e" | "f"

Ostring ::= "'" { [Oct](#TOct) | BinSpace } "'" "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

 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)] [Type](#TType) [Identifier](#TIdentifier)

 [":=" ([Expression](#TExpression) | [Minus](#TMinus))]

FormalPortPar ::= [[InOutParKeyword](#TInOutParKeyword)] [Identifier](#TIdentifier) [Identifier](#TIdentifier)

/\* The first Identifier refers to the port type. The second Identifier refers to the port parameter identifier \*/

FormalTimerPar ::= [[InOutParKeyword](#TInOutParKeyword)] [TimerKeyword](#TTimerKeyword) [Identifier](#TIdentifier)

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

 [OutParKeyword](#TOutParKeyword) |

 [InOutParKeyword](#TInOutParKeyword)

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

 [FuzzyModifier](#TFuzzyModifier)]

 [Type](#TType) [Identifier](#TIdentifier) [":=" (TemplateInstance | [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 ::= [AttribKeyword](#TAttribKeyword) [[OverrideKeyword](#TOverrideKeyword)] [[AttribQualifier](#TAttribQualifier)]

 [FreeText](#TFreeText)

AttribKeyword ::= [EncodeKeyword](#TEncodeKeyword) |

 [VariantKeyword](#TVariantKeyword) |

 [DisplayKeyword](#TDisplayKeyword) |

 [ExtensionKeyword](#TExtensionKeyword) |

 [OptionalKeyword](#TOptionalKeyword)

EncodeKeyword ::= "encode"

VariantKeyword ::= "variant"

DisplayKeyword ::= "display"

ExtensionKeyword ::= "extension"

OverrideKeyword ::= "override"

AttribQualifier ::= "(" [DefOrFieldRefList](#TDefOrFieldRefList) ")"

DefOrFieldRefList ::= [DefOrFieldRef](#TDefOrFieldRef) {"," [DefOrFieldRef](#TDefOrFieldRef)}

DefOrFieldRef ::= [QualifiedIdentifier](#TQualifiedIdentifier) |

 (([FieldReference](#TFieldReference) | "[" [Minus](#TMinus) "]") [[ExtendedFieldReference](#TExtendedFieldReference)]) |

 [AllRef](#TAllRef)

QualifiedIdentifier ::= {[Identifier](#TIdentifier) [Dot](#TDot)} [Identifier](#TIdentifier)

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]

/\* 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) ["(" [ComponentOrDefaultReference](#TComponentOrDefaultReference)

 ")"]

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) | [ArrayExpression](#TArrayExpression)

/\* STATIC SEMANTICS - Within CompoundExpression the ArrayExpression can be used for Arrays, record, record of and set of types. \*/

FieldExpressionList ::= "{" [FieldExpressionSpec](#TFieldExpressionSpec) {"," [FieldExpressionSpec](#TFieldExpressionSpec)}

 "}"

FieldExpressionSpec ::= [FieldReference](#TFieldReference) [AssignmentChar](#TAssignmentChar) [NotUsedOrExpression](#TNotUsedOrExpression)

ArrayExpression ::= "{" [[ArrayElementExpressionList](#TArrayElementExpressionList)] "}"

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 ::= [VariableRef](#TVariableRef) [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) |

 ("[" [Minus](#TMinus) "]")

 }+

/\* STATIC SEMANTIC - The Identifier refers to a type definition if the type of the VarInstance or ReferencedValue in which the ExtendedFieldReference is used is anytype. ArrayOrBitRef shall be used when referencing elements of values or arrays. The square brackets with dash shall be used when referencing inner types of a record of or set of type. \*/

OpCall ::= [ConfigurationOps](#TConfigurationOps) |

 [GetLocalVerdict](#TGetLocalVerdict) |

 [TimerOps](#TTimerOps) |

 [TestcaseInstance](#TTestcaseInstance) |

 ([FunctionInstance](#TFunctionInstance) [[ExtendedFieldReference](#TExtendedFieldReference)]) |

 ([TemplateOps](#TTemplateOps) [[ExtendedFieldReference](#TExtendedFieldReference)]) |

 [ActivateOp](#TActivateOp)

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

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] "}"

SelectCase ::= [CaseKeyword](#TCaseKeyword) ("(" TemplateInstance {"," TemplateInstance }

 ")" | [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 ElseKeyword StatementBlock

CaseKeyword ::= "case"

ExtendedIdentifier ::= [[Identifier](#TIdentifier) [Dot](#TDot)] [Identifier](#TIdentifier)

/\*\* STATIC SEMANTICS The optional Identifier Dot part shall not be used for enumerated values\*/

IdentifierList ::= [Identifier](#TIdentifier) {"," [Identifier](#TIdentifier)}

QualifiedIdentifierList ::= [QualifiedIdentifier](#TQualifiedIdentifier) {"," [QualifiedIdentifier](#TQualifiedIdentifier)}

### A.1.6.9 Miscellaneous productions

Dot ::= "."

Minus ::= "-"

SemiColon ::= ";"

Colon ::= ":"

Underscore ::= "\_"

AssignmentChar ::= ":="

IndexModifier ::= "@index"

DeterministicModifier ::= "@deterministic"

LazyModifier ::= "@lazy"

FuzzyModifier ::= "@fuzzy"

CaseInsenModifier ::= "@nocase"

DecodedModifier ::= "@decoded"