### 16.1.4 Invoking functions from specific places

If value returning functions are called in receiving communication operations (in templates, template fields, in-line templates, or as actual parameters), in guards or events of alt statements or altsteps (see clause 20.2), or in initializations of altstep local definitions (see clause 16.2), the following operations shall not be used in functions called in the cases specified above, in order to avoid side effects that cause changing the state of the component or the actual snapshot and to prevent different results of subsequent evaluations on an unchanged snapshot:

1. All component operations, i.e. **create**, **start**(component), **stop**(component), **kill**, **running**(component), **alive,** **done** and **killed** (see notes 1, 3, 4 and 6).
2. All port operations, i.e. **start**(port), **stop**(port), **halt**, **clear**, **checkstate**, **send**, **receive**, **trigger**, **call**, **getcall**, **reply**, **getreply**, **raise**, **catch**, **check**, **connect**, **disconnect**, **map** and **unmap** (see notes 1, 2, 3, 4 and 6).
3. The **action** operation (see notes 2 and 6).
4. All timer operations, i.e. **start**(timer), **stop**(timer), **running**(timer), **read**, **timeout** (see notes 4 and 6).
5. Calling non-deterministic external functions, i.e. external functions where the resulting values for actual inout or out parameters or the return value may differ for different invocations with the same actual in and inout parameters (see notes 4 and 6).
6. Calling the **rnd** predefined function (see notes 4 and 6).
7. Changing of component variables, i.e. using component variables on the left-hand side of assignments, and in the instantiation of **out** and **inout** parameters (see notes 4 and 6).
8. Calling the **setverdict** operation (see notes 4 and 6).
9. Activation and deactivation of defaults, i.e. the **activate** and **deactivate** statements (see notes 5 and 6).
10. Calling functions and deterministic external functions with **out** or **inout** parameters (see notes 7 and 8).
11. The **encode** operation (see note 8)

NOTE 1: The execution of the operations **start**, **stop**, **done**, **killed**, **halt**, **clear**, **receive**, **trigger**, **getcall**, **getreply**, **catch** and **check** can cause changes to the current snapshot.

NOTE 2: The use of operations **send**, **call**, **reply**, **raise**, and **action** causes an error, i.e. all communication are to be made explicit and not as a side effect of another communication operation or the evaluation of a snapshot.

NOTE 3: The use of operations **map**, **unmap**, **connect**, **disconnect**, **create** causes an error, i.e. all configuration operations are to be made explicit, and not as a side effect of a communication operation or the evaluation of a snapshot.

NOTE 4: Calling of non-deterministic external functions, **rnd**, **running**, **alive**, **read**, **checkstate**, **setverdict**, and writing to component variables causes an error because this may lead to different results of subsequent evaluations of the same snapshot, thus, e.g. rendering deadlock detection impossible.

NOTE 5: The use of operations **activate** and **deactivate** causes an error because they modify the set of defaults that is considered during the evaluation of the current snapshot.

NOTE 6: Restrictions except the limitation on the use of **out** or **inout** parameterization in restriction j) apply recursively, i.e. it is disallowed to use them directly, or via an arbitrary long chain of function invocations.

NOTE 7: The restriction of calling functions and deterministic external functions with **out** or **inout** parameters does not apply recursively, i.e. calling functions that themselves call functions with **out** or **inout** parameters is legal.

NOTE 8: Using **out** or **inout** parameters and the **encode** operation causes an error because this may lead to different results of subsequent evaluations of the same snapshot.

# 27 Specifying attributes

## 27.0 General

TTCN‑3 uses attributes to give special characterization/meaning to language elements such as specific presentation format, specific encoding and encoding variants, and user-defined properties.

## 27.1 The Attribute mechanism

### 27.1.0 General

Attributes can be associated with TTCN‑3 language elements by means of the **with** statement. The **with** statement can be applied to modules, global module definitions and to local definitions in control, test cases, functions, altsteps, statement blocks and in component type definitions.

### 27.1.1 Scope of attributes

A **with** statement may associate attributes to a single language element or to elements or fields of structured types (in a recursive way) or to members of component or port types, the same way as specified in clauses 6.2.1.1 and 6.2.3.2. It is also possible to associate attributes to a number oflanguage elements by, e.g. listing fields of a structured type in an attribute statement associated with a single type definition or associating a **with** statement to the surrounding scope unit or **group** of language elements.

A **with** statement can follow any module, any global definition inside module and group declarations as well as any local definition in component types and statement blocks inside behaviour definitions or the control part.

EXAMPLE 1: // attributes for single language elements and groups

 // MyPDU1 will be displayed as PDU

 **type** **record** MyPDU1 { … } **with** { **display** "PDU"}

 // MyPDU2 will be displayed as PDU with the application specific extension attribute MyRule

 **type** **record** MyPDU2 { … }

 **with**

{

 **display** "PDU";

 **extension** "MyRule"

 }

 // The following group definition …

 **group** myPDUs {

 **type** **record** MyPDU3 { … }

 **type** **record** MyPDU4 { … }

 }

 **with** {**display** "PDU"} // All types of group MyPDUs will be displayed as PDU

 // is identical to

 **group** myPDUs {

 **type** **record** MyPDU3 { … } **with** { **display** "PDU"}

 **type** **record** MyPDU4 { … } **with** { **display** "PDU"}

 }

EXAMPLE 2: // attributes for fields and elements

 **type** **record** MyRec {

 **integer** field1,

 **record** {

 **integer** eField1,

 **boolean** eField2

 } field2

 }

 **with** { **display** (field2.eField1) "colour blue" }

 // the embedded field eField1 is displayed blue

 **type** **record of integer** MyRecOfInteger

 **with** { **display** ([-]) "colour green"

 // all integer elements are displayed green

 **type** **record** of **integer** MyRecOfInteger2

 **with** { **display** ([-]) "colour red" }

 // integer elements are displayed red

 **const** MyRecOfInteger c\_MyRecordOfInt := {0, 1, 2, 3}

 **with** { **display** ([0]) "colour blue" }

 // the first element is displayed blue, the other elements are displayed red

### 27.1.2 Overwriting rules for attributes

#### 27.1.2.0 General

An attribute definition in a lower scope unit will override a general attribute definition in a higher scope and a type-specific attribute inherited from a type reference. Attributes inherited from a type reference will override general attributes from a higher scope unit. Additional overwriting rules for variant attributes are defined in clause 27.1.2.1.

EXAMPLE 1:

 **type** **record** MyRecordA

 {

 :

 } **with** { **encode** "RuleA" }

 // In the following, MyRecordA is encoded according to "RuleA" and not according to
 // "RuleB" because the attribute from the referenced type MyRecordA overrides

 // the attribute from higher scope unit (surrounding MyRecordB type).

 **type record** MyRecordB

 {

 :

 MyRecordA field

 } **with** { **encode** "RuleB" }

A **with** statement that is placed inside the scope of another **with** statement shall override the outermost **with**. This shall also apply to the use of the **with** statement with groups. If multiple attributes of the same type are allowed, all of them are overridden unless specified otherwise.

EXAMPLE 2:

 // Example of the use ofthe overwriting scheme of the **with** statement

 **group** myPDUs

 {

 **type** **record** MyPDU1 { … }

 **type** **record** MyPDU2 { … }

 **group** mySpecialPDUs

 {

 **type** **record** MyPDU3 { … }

 **type** **record** MyPDU4 { … }

 }

 **with** {**extension** "MySpecialRule"} // MyPDU3 and MyPDU4 will have the application

 // specific extension attribute MySpecialRule

 }

 **with**

{

 **display** "PDU"; // All types of group myPDUs will be displayed as PDU and

 **extension** "MyRule"; // (if not overwritten) have the extension attribute MyRule

 }

 // is identical to …

 **group** myPDUs

 {

 **type** **record** MyPDU1 { … } **with** {**display** "PDU"; **extension** "MyRule" }

 **type** **record** MyPDU2 { … } **with** {**display** "PDU"; **extension** "MyRule" }

  **group** mySpecialPDUs {

 **type** **record** MyPDU3 { … } **with** {**display** "PDU"; **extension** "MySpecialRule" }

 **type** **record** MyPDU4 { … } **with** {**display** "PDU"; **extension** "MySpecialRule" }

 }

 }

Attributes with **@local** modifier **o**verride attributes from higher scope, but they are valid for the associated language element only. They do not affect definitions inside the associated language element as the **@local** attribute is completely transparent to lower scopes. Attributes from higher scope will still affect attributes in lower scopes even if the **@local** attribute is between them.

EXAMPLE 3:

 module M {

 **type** **record** MyRec {

 **integer** field1,

 **integer** field1,

 } **with** { **encode @local** "CodecB" }

 // the record type MyRec will be encoded with CodecB, but its fields with CodecA,

 // because local attribute CodecB doesn’t affect elements of the MyRec type.

 } **with** { **encode** "CodecA" }

An attribute definition in a lower scope can be overwritten in a higher scope by using the **override** directive.

EXAMPLE 4:

 **type** **record** MyRecordA

 {

 :

 } **with** { **encode** "RuleA" }

 // In the following, fieldA of a MyRecordB instance is encoded according to RuleB

 **type record** MyRecordB

 {

 :

 MyRecordA fieldA

 } **with** { **encode** **override** (fieldA) "RuleB" }

The **override** directive forces all contained types at all lower scopes to be forced to the specified attribute. If applied to a type, the directive does not override the specified attribute in language elements that reference this type.

An attribute definition for a field or element of a structured type overrides the corresponding attribute of the structured type, as regards the identified field or element. The attribute definition for a field or element of a structured type can however be overridden with the override directive in the attribute definition of the structured type.

In case of nested override directives, the override directive of the highest scope shall take precedence. Override directives of definitions at the same scope level do not effect each other.

EXAMPLE 5:

 // An instance of MyRecordA is encoded according to RuleA.

 **type** **record** MyRecordA

 {

 :

 } **with** { **encode** **override** "RuleA" }

 // In the following, fieldA of a MyRecordB instance (and all its sub-fields) is encoded

 // according to "RuleB".

 **type** **record** MyRecordB

 {

 :

 MyRecordA fieldA

 } **with** { **encode** **override** "RuleB" }

 // In the following, both rules "RuleA" and "RuleB" are overriden by "RuleC" of the group.

 **group** myGroup {

 **type** **record** MyRecordA

 {

 :

 } **with** { **encode** **override** "RuleA" }

 **type** **record** MyRecordB

 {

 :

 MyRecordA fieldA

 } **with** { **encode** **override** (fieldA) "RuleB" }

 } **with** { **encode** **override** "RuleC" }

 // In the following, the template mw\_msg will be encoded with "RuleB", because the

 // override directive doesn’t override the encode attribute in references. However,

 // all fields of the mw\_template will be encoded with "RuleA", because the attributes

 // from the references have higher precedence than attributes from a higher scope.

 **type** **record** MyRecordC

 {

 :

 } **with** { **encode** **override** "RuleA" }

 **template MyRecordC mw\_msg :=**

 {

 :

 } **with** { **encode** "RuleB" }

#### 27.1.2.1 Additional default overwriting rules for variant attributes

A **variant** attribute is always related to an **encode** attribute. Whereas a variant of an encoding may change, an encoding shall not change without overwriting all current variant attributes.

The present document defines the default rules for variant attributes. Extension packages of TTCN-3, for example specifying language mappings, may define their own overwriting rules for variant attributes. For variant attributes the following default overwriting rules apply:

* a **variant** attribute overwrites an current **variant** attribute according to the rules defined in clause 27.1.2;
* an **encoding** attribute, which overwrites a current **encoding** attribute according to the rules defined in clause 27.1.2, also overwrites a corresponding current **variant** attribute, i.e. no new **variant** attribute is provided, but the current **variant** attribute becomes inactive;
* an **encoding** attribute, which changes a current **encoding** attribute of an imported language element according to the rules defined in clause 27.1.3, also changes a corresponding current **variant** attribute, i.e. no new **variant** attribute is provided, but the current **variant** attribute becomes inactive.

EXAMPLE:

 **module** MyVariantEncodingModule {

 :

 **type** **charstring** MyType**;** // Normally encoded according to "Encoding 1"

 :

 **group** myVariantsOne {

 :

 **type record** MyPDUone

 {

 **integer** field1, // field1 will be encoded according to "Encoding 2" only.

 // "Encoding 2" overwrites "Encoding 1" and variant "Variant 1"

 MyType field3 // field3 will be encoded according to "Encoding 1" with

 // variant "Variant 1".

 }

 **with** { **encoding** (field1) "Encoding 2" }

 :

 }

 **with** { **variant** "Variant 1" }

 **group** myVariantsTwo

 { :

 **type record** MyPDUtwo

 {

 **integer** field1, // field1 will be encoded according to "Encoding 3"

 // using encoding variant "Variant 3"

 MyType field3 // field3 will be encoded according to "Encoding 1"

 // using encoding variant "Variant 2"

 }

 **with** { **variant** (field1) "Variant 3" }

 :

 }

 **with** { **encode** "Encoding 3"; **variant** "Variant 2"}

 }

 **with** { **encode** "Encoding 1" }

#### 27.1.2.2 Overwriting rules for multiple encoding

Explicitly listed encode attributes that occur on the higher scope and are not overwritten will retain all variants related to them. All variants that do not contain an encode reference will be also kept in this case unless they are explicitly overwritten in the **with** statement.

A variant with an encoding reference will overwrite only variants with the same encoding reference. A variant with no encoding reference will overwrite only variants with no encoding reference.

 An encode attribute with an **@only** modifier is used to select a single encode attribute from a list of multiple encode attributes the language element inherits from a higher scope or type reference. Before applying the **@only** attribute, encode attribute values are resolved as if the the **@only** attribute were missing. If this procedure produces multiple encode attributes and one of those is equal to the value of the **@only** attribute, the **@only** clause shortens the list of encode attributes to this single value. All remaining encode attributes and variants associated with them shall be discarded. The procedure is recursively applied to all fields, elements and lower scopes of the associated language element. If there are zero or one encode attributes or if the list of encode attributes doesn’t contain the value of the **@only** attribute, the @only attribute is ignored.

EXAMPLE:

 **type integer** Int **with** {

 **encode** "CodecA"; **variant** "CodecA"."Rule1";

 **encode** "CodecB"; **variant** "CodecB"."Rule2";

 **variant** "GlobalRule"; // Global variant

 }

 // Modifying list of allowed encodings

 type Int Int2 **with** {

 **encode** "CodecA"; // variants "CodecA"."Rule1" and "GlobalRule" are kept

 **encode** "CodecC"; **variant** "CodecC"."Rule6"; // new encoding and related variant

 // "CodecB" encoding and related variant are discarded as "CodecB" is not

 // explicitly referenced

 }

 // Overwriting variant with an encoding reference

 type Int Int3 **with** {

 **variant** "CodecB"."Rule4"; // new variant for encoding "CodecB" overwrite

 // the original variant "CodecB"."Rule2"

 // Variants "CodecA"."Rule1" and "GlobalRule" are unchanged as they

 // contain no reference to "CodecB"

 }

 **type record** MyRecord {

 **integer** field1,

 **integer** field2

 } **with** {

 **encode** "CodecA"; **variant** "CodecA"."Rule1";

 **encode** "CodecB"; **variant** "CodecB"."Rule2";

 **variant** "GlobalRule"; // Global variant

 }

 // Selecting encoding with an @only modifier

 type MyRecord MyRecord2 **with** {

 **encode @only** (field1)"CodecB"; // field1 will be encoded with CodecB

 **encode @only** "CodecA" // the record and field2 will be encoded with CodecA

 }

### 27.1.3 Changing attributes of imported language elements

In general, a language element is imported together with its attributes. In some cases these attributes may have to be changed when importing the language element, e.g. a type may be displayed in one module as ASP, then it is imported by another module where it should be displayed as PDU. For such cases it is allowed to change attributes on the **import** statement.

When resolving the attributes, the **import** statement works as an additional higher scope unit on the top of the imported module. Attributes set in the import statement are valid only within the importing module.

NOTE 1: The import statement occurs inside an importing module and sometimes inside a group. Because of the scope rules, attributes of these scope units apply to the imported module too.

NOTE 2: If a **with** statement is added to an import of a definition where a local definition also has a **with** statement, the local definition's attributes overwrite the attributes added to the import statement in the normal way. Thus, if the attributes of a local definition shall be changed via the import statement, the override directive needs to be used.

EXAMPLE:

 **import from** MyModule {

 **type** MyType

 }

 **with** { **display** "ASP" } // MyType will be displayed as ASP

 **import from** MyModule {

 **group** myGroup

 }

 **with** {

 **display** "PDU"; // By default all types will be displayed as PDU

 **extension** "MyRule"

 }

## 27.2 The With statement

The with statement is used to associate attributes to TTCN‑3 language elements (and sets thereof).

***Syntactical Structure***

**with** "{"

{ ( **encode** | **variant** | **display** | **extension** | **optional** )

 [ **override** | @local | @only ]

 ["(" *DefinitionRef* | *FieldReference* | *AllRef* ")"]

 *FreeText* [ "." *FreeText* ] [";"] }

"}"

***Semantic Description***

There are five kinds of attributes that can be associated to language elements:

a) **display:** allows the specification of display attributes related to specific presentation formats;

b) **encode:** allows references to specific encoding rules;

c) **variant:** allows references to specific encoding variants;

d) **extension:** allows the specification of user-defined attributes;

e) **optional:** allows the implicit setting of optional fields in records and sets to omit.

The syntax for the argument of the **with** statement (i.e. the actual attributes) is defined as a free text string.

*DefinitionRef* and *FieldReference* identify a definition or field respectively which is within the module, group or definition to which the **with** statement is associated.

*AllRef* can be used to apply attributes to multiple language elements within the scope to which the **with** statement is associated. *AllRef* provides a flexible mechanism to select all language elements or all language elements of a certain kind in a given scope. Individual language elements that are not affected by an attribute can be excluded from a set of selected language elements.

***Restrictions***

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

a) *DefinitionRef* and *FieldReference* shall refer to a definition or field respectively which is within the module, group or definition to which the **with** statement is associated.

b) *AllRef* shall identify a multiple language elements within the scope unit to which the with statement is associated

c) The **@only** attribute modifier shall be used only for encode attributes

d) In case multiple attributes of the same type are allowed, all of them shall be without an additional modifier (**override**, **@local**, **@only**) or the modifier shall be the same for all attributes.

e) Dot notation in the *FreeText* part is allowed for variant attributes only.

***Examples***

 **type** **record** MyService {

 **integer** i,

 **float** f

 }

 **with** { **display** "ServiceCall" } // MyRecord will be displayed as a ServiceCall

## 27.3 Display attributes

Display attributes allow the specification of display attributes related to specific presentation formats.

***Syntactical Structure***

**display**

***Semantic Description***

All TTCN‑3 language elements can have **display** attributes to specify how particular language elements shall be displayed in, for example, a tabular format.

Special attribute strings related to the display attributes for the graphical presentation format can be found in ETSI ES 201 873-3 [i.2].

Other **display** attributes may be defined by the user.

NOTE: Because user-defined attributes are not standardized, the interpretation of these attributes may differ between tools or even may not be supported.

***Restrictions***

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

1. At most one display attribute shall be applied to each definition, each individual field reference or language element to which a **with** statement is associated.

***Examples***

 **type** **record** MyService {

 **integer** i,

 **float** f

 }

 **with** { **display** "ServiceCall" } // MyRecord will be displayed as a ServiceCall

## 27.4 Encoding attributes

In TTCN‑3, general or particular encoding rules can be specified by using **encode** and **variant** attributes. Encoding attributes allow references to specific encoding rules.

***Syntactical Structure***

**encode**

***Semantic Description***

Encoding rules define how a particular value, template, etc. shall be encoded and transmitted over a communication **port** and how received signals shall be decoded. TTCN‑3 does not have a default encoding mechanism. This means that encoding rules or encoding directives are defined in some external manner to TTCN‑3.

The **encode** attribute allows the association of some referenced encoding rule or encoding directive to be made to a TTCN‑3 definition.

The manner in which the actual encoding rules are defined (e.g. prose, functions, etc.) is outside the scope of the present document. If no specific rules are referenced then encoding shall be a matter for individual implementation.

In most cases encoding attributes will be used in a hierarchical manner. The top-level is the entiremodule, the next level is a group and the lowest is an individual type or definition:

a) **module:** encoding applies to all types defined in the module, including TTCN‑3 types (built-in types);

b) **group:** encoding applies to a group of user-defined type definitions;

c) **type or definition:** encoding applies to a single user-defined type or definition;

d) **field:** encoding applies to a field in a **record** or **set** type or **template**.

The **with** statement can contain more than one encode attribute. In this case, the multiple encodings are supported in the context where the attribute is used. The encoding used in the encoding and decoding operations can be selected statically the **@only** attribute modifier (27.1.2.2), dynamically using the encode operation (27.9.) and as a parameter of predefined codec functions (C.5) or inside the codec implementation.

***Restrictions***

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

a)If a single language element has an encode attribute with an **@only** modifier associated with it, no other encoding attributes are allowed for this language elements.

***Examples***

 **module** MyFirstmodule

 { :

 **import from** MySecondModule {

 **type** MyRecord

 }

 **with** { **encode** "MyRule 1" } // Instances of MyRecord will be encoded according to MyRule 1

 :

 **type** **charstring** MyType**;** // Normally encoded according to the 'Global encoding rule

 :

 **group** myRecords

 { :

 **type record** MyPDU1

 {

 **integer** field1, // field1 will be encoded according to "Rule 3"

 **boolean** field2**,** // field2 will be encoded according to "Rule 3"

 Mytype field3 // field3 will be encoded according to "Rule 2"

 }

 **with** { **encode** (field1, field2) "Rule 3" }

 :

 }

 **with** { **encode** "Rule 2" }

 }

 **with** { **encode** "Global encoding rule" }

## 27.5 Variant attributes

In TTCN‑3, general or particular encoding rules can be specified by using **encode** and **variant** attributes. Variant attributes allow references to specific encoding variants.

***Syntactical Structure***

**variant**

***Semantic Description***

To specify a refinement of the currently specified encoding scheme instead of its replacement, the **variant** attribute shall be used. The variant attributes are different from other attributes, because they are closely related to encode attributes. Therefore, for variant attributes, additional overwriting rules apply (see clause 27.1.2.1).

**Special variant strings:**

The following strings are the predefined (standardized) **variant** attributes for simple basic types (see clause E.2.1):

a) "8 bit" and "unsigned 8 bit" mean, when applied to integer and enumerated types, that the integer value or the integer numbers associated with enumerated values shall be handled as it was represented on 8‑bits (single byte) within the system.

b) "16 bit" and "unsigned 16 bit" mean, when applied to integer and enumerated types, that the integer value or the integer numbers associated with enumerated values shall be handled as it was represented on 16‑bits (two bytes) within the system.

c) "32 bit" and "unsigned 32 bit" mean, when applied to integer and enumerated types, that the integer value or the integer numbers associated with enumerated values shall be handled as it was represented on 32‑bits (four bytes) within the system.

d) "64 bit" and "unsigned 64 bit" mean, when applied to integer and enumerated types, that the integer value or the integer numbers associated with enumerated values shall be handled as it was represented on 64‑bits (eight bytes) within the system.

e) "IEEE754 float","IEEE754 double", "IEEE754 extended float" and "IEEE754 extended double" mean, when applied to a float type, that the value shall be encoded and decoded according to the standard IEEE™ 754 [6] (see annex E).

The following strings are the predefined (standardized) **variant** attributes for **charstring** and **universal charstring** (see clause E.2.2):

a) "UTF-8" means, when applied to the universal charstring type, that the value shall be encoded and decoded according to the UCS encoding scheme UTF-8 as defined in clause 10.1 of ISO/IEC 10646 [2].

b) "UTF-16" means, when applied to the universal charstring type, that the value shall be encoded and decoded according to the UCS encoding scheme UTF-16 as defined in clause 10.4 of ISO/IEC 10646 [2].

c) "UTF-16LE" means, when applied to the universal charstring type, that each character of the value shall be individually encoded and decoded according to the UCS Encoding scheme UTF-16LE as defined in clause 10.3 of ISO/IEC 10646 [2].

d) "UTF-16BE" means, when applied to the universal charstring type, that the value shall be encoded and decoded according to the UCS Encoding scheme UTF-16BE as defined in clause 10.2 of ISO/IEC 10646 [2].

e) "UTF-32" means, when applied to the universal charstring type, that the value shall be encoded and decoded according to the UCS Encoding scheme UTF-32 as defined in clause 10.7 of ISO/IEC 10646 [2].

f) "UTF-32LE" means, when applied to the universal charstring type, that the value shall be encoded and decoded according to the UCS Encoding scheme UTF-32LE as defined in clause 10.6 of ISO/IEC 10646 [2].

g) "UTF-32BE" means, when applied to the universal charstring type, that the value shall be encoded and decoded according to the UCS Encoding scheme UTF-32BE as defined in clause 10.5 of ISO/IEC 10646 [2].

h) "8 bit" means, when applied to charstring and universal charstring types, that each character of the value shall be individually encoded and decoded according to the coded representation as specified in ISO/IEC 10646 [2] (an 8-bit coding).

NOTE: The UCS Encoding schemes allow an optional signature (also known as byte order mark, BOM) to be present in encoded character strings. The above UCS encoding scheme variant attributes does not specify, if signatures are present in the encoded values or not, this is an option for the encoder. It is expected that decoders are able to process signatures in the decoding process.

The following strings are the predefined (standardized) **variant** attributes for structured types (see clause E.2.2.4):

a) "IDL:fixed FORMAL/01-12-01 v.2.6" means, when applied to a record type, that the value shall be handled as an IDL fixed point decimal value (see annex E).

These variant attributes can be used in combination with the more general encode attributes specified at a higher level. For example a **universal charstring** specified with the **variant** attribute "UTF-8" within a module which itself has a global encoding attribute "BER:1997" (see clause 12.2 of ETSI ES 201 873-7 [i.5]) will cause each character of the values within the string to first be encoded following the UTF-8 rules and then this UTF-8 value will be encoded following the more global BER rules.

**Invalid encodings**

If it is desired to specify invalid encoding rules then these shall be specified in a referenceable source external to the module in the same way that valid encoding rules are referenced.

**Multiple encodings**

In case multiple encodings (specified in 27.4) are used, the variant attribute value can be composed of two strings separated by a dot. The first string defines the encoding the variant is related to. Such an attribute is valid only when the related encoding is selected. It is still possible to specify variants without encoding attribute reference when multiple encodings are used. These attributes apply to all defined encodings.

**Multiple variants**

The **with** statement can contain any number of variant attributes.

***Restrictions***

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

a)When dot notation is used in the variant attribute value, the first string shall resolve into a valid encoding attribute value.

***Examples***

 **module** MyTTCNmodule1

 { :

 **type** **charstring** MyType**;** // Normally encoded according to the "Global encoding rule"

 :

 **group** myRecords

 { :

 **type record** MyPDU1

 {

 **integer** field1, // field1 will be encoded according to "Rule 2"

 // using encoding variant "length form 3"

 Mytype field3 // field3 will be encoded according to "Rule 2"

 // using any possible length encoding format

 }

 **with** { **variant** (field1) "length form 3" }

 :

 }

 **with** { **encode** "Rule 2" }

 **type** **charstring** Multi **with** {

 **encode** "Codec1"; **variant** "Codec1"."Rule1";

 **encode** "Codec2"; **variant** "Codec2"."Rule3";

 }**; // multiple encodings (**"**Codec1**"**,** "**Codec2**"**), the first variant** "Rule1" **is valid**

 **// only for the** "**Codec1**" encoding, the second variant "Rule3" is valid for

 // the "**Codec2**" encoding

 }

 **with** { **encode** "Global encoding rule" }

## 27.6 Extension attributes

Extension attributes can be used for proprietary extensions to TTCN‑3. The **with** statement can contain any number of extension attributes.

***Syntactical Structure***

**extension**

***Semantic Description***

All TTCN‑3 language elements can have **extension** attributes specified by the user.

NOTE: Because user-defined attributes are not standardized the interpretation of these attributes between tools supplied by different vendors may differ or even not be supported.

***Restrictions***

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

***Examples***

 **testcase** TC\_MyTestcase() **runs** **on** MTCType {

 :

 }

 **with** { **extension** "Test Purpose: This test case is used to check …" }

## 27.7 Optional attributes

The **optional** attribute can be used to indicate that optional fields of constants, module parameters or templates of record and set types are implicitly set to **omit**.

***Syntactical Structure***

**optional**

***Semantic Description***

TTCN‑3 constants, module parameters, and templates can have an **optional** attribute. Also, TTCN-3 language elements that contain such definitions, i.e. module, group, function, altstep, test case, control, and component type definitions can have an **optional** attribute. When an **optional** attribute is associated to a function, altstep, test case, control or component type definitions, it shall have effect on all the constants, module parameters, and templates declared within these definitions and not on the enframing definition itself.

**Special optional strings:**

The following strings are the predefined (standardized) **optional** attributes.

a) "implicit omit" means that all optional fields, that have no assigned value definition in the statement on which the attribute operates, are set to omit. This applies recursively to the optional fields of the entity and to subfields of the mandatory fields.

b) "explicit omit" means that all optional fields, that have no assigned value definition in the statement on which the attribute operates, are left undefined. This applies recursively to the optional fields of the entity and to subfields of the mandatory fields.

***Restrictions***

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

1. Data type, port type, procedure signature and variable definitions and import statements shall not have an **optional** attribute associated to them directly. When an **optional** attribute is associated to module, group, function, altstep, test case, control or component type containing such definitions, it shall not have any effect on the included data type, port type, procedure signature, variable or import statement.
2. At most one optional attribute shall be applied to each definition, each individual field reference or language element to which a **with** statement is associated.

***Examples***

**type** **record** MyRecord1 { **integer** a, **boolean** b **optional**}
**type record** MyRecord2{MyRecord1m}
// reference templates with explicitly set fields
**template** MyRecord1mw\_myTemplate1:= { a := ?, b := **omit** }
**template** MyRecord2mw\_myTemplate2:= { m := { a := ?, b := **omit** }}// reference templates
**template** MyRecord1mw\_myTemplate1a:= {a := ? } // b is undefined
**template** MyRecord1mw\_myTemplate1b:= {a := ? } **with** {**optional** "explicit omit**"**} // b is undefined
 **template** MyRecord2mw\_myTemplate2a:= {} // m and its subfields are undefined

**template** MyRecord2mw\_myTemplate2b:= { m := { a := ?}}; // m.b is undefined

// templates with attribute

 **template** MyRecord1mw\_myTemplate11 **:=** { a := ? } **with** {**optional "**implicit omit**"**}
 // same as mw\_myTemplate1, b is set to omit

**template** MyRecord2mw\_myTemplate21:= { m := { a := ?}} **with** {**optional "**implicit omit**"**}
// same as mw\_myTemplate2, by recursive application of the attribute

**template** MyRecord2mw\_myTemplate22:= { m := mw\_myTemplate1a } **with** {**optional "**implicit omit**"**}
// same as mw\_myTemplate2, by recursive application of the attribute

**template** MyRecord2mw\_myTemplate23:= {} **with** {**optional "**implicit omit**"**}
 // same as mw\_myTemplate2a, m remains undefined

**template** MyRecord2mw\_myTemplate24 **:=** { m := mw\_myTemplate1b } **with** {**optional "**implicit omit**"**}
// same as mw\_myTemplate2b, the attribute on the lower scope is not overwritten

**template** MyRecord2mw\_myTemplate25:= { m := MyTemplate1b } **with** {**optional override "**implicit omit**"**}
 // same as mw\_myTemplate2, the attribute on the lower scope is overwritten

## 27.9 Dynamic selection of encode attributes

The **encode** operation can be used to dynamically select a single encoding attribute of a type that supports multiple encodings.

***Syntactical Structure***

( *Port* | ( **all** **port** ) | **self** ) "." **encode** "(" *Type* "," ( *SingleExpression* | - ) ")"

**optional**

***Semantic Description***

The **encode** operation dynamically restricts the number of encode attributes of a referenced type or its fields or elements to a single value. Dependent on the language element preceding the dot, encode attribute selection is valid either for sending and receiving operations of a single port (single port reference), sending and receiving operations of all ports of the current component (**all** **port** notation) or for all codec function and communication operation of the current component (**self** keyword).

Rules for dynamic encoding selection are similar to the rules for **@only** attribute modifier. If the referenced type contains multiple encode attributes and the expression provided in the **encode** operation is equal to one of these encode attribute values, the statement reduces the list of encode attributes to the selected one. The procedure is applied recursively to all elements and fields or the referenced type. After executing the operation, all other encode attributes and variants related to them are dynamically disabled and invisible to the codec.

The second parameter of the **encode** operation may contain the not used symbol “-“ instead of an expression. In this case, the operation removes dynamic encoding selection applied to the type by a preceding **encode** operation call and restores the original multiple encode attribute.

Repeated call of the **encode** operation always use static attributes that are valid for the referenced type. Previous calls of the encode operation referencing the type are not considered in this case. This way it is possible to change the encoding during test execution without having to reset it first.

It is allowed to reference a field or element of a type using an extended type reference in the **encode** operation. This operation is useful for payload fields of container protocols and allows dynamical selection of proper encoding for payload fields. If the extended type reference is used, following call of the encode operation for the whole type won’t change the encoding that was dynamically selected for this field or element (and its sub‑fields).

***Restrictions***

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

1. If the **encode** operation is applied to a single port, the referenced type shall be either listed in the **in** or **out** type list of the related port type or it shall be a reference to a field or element of a type listed in the **in** or **out** type list of the related port type.
2. The *SingleExpression* used in the second parameter of the encode operation shall be of a universal charstring type.

***Examples***

**type port** P **message** {
 **inout** PDU;
} **type type** **component** C {
 **port** P p;
} **// Payload type with two encoding options
type** **record** Payload { **...**} **with** { **encode** "PayloadCodec1"; **encode** "PayloadCodec2" }
 **// PDU type with two encoding options
type record** PDU{charstring source,
charstring destination,
Payload payload} **with** { **encode** "PduCodec1"; **encode** "PduCodec2" }

template PDU m\_msg := {
 source := "source address",
 destination := "destination address",
 payload := { ... }
}

testcase TC01() runs on C {
 p.encode(ContainerType.payload, "PayloadCodec2");
 p.encode(ContainerType, "PduCodec1");
 p.send(m\_msg); // m\_msg will be sent with encode attribute "PduCodec1" and its payload
 // field will have encode attribute "PayloadCodec2"
 p.encode(ContainerType, -); // resets encoding of the PDU to the original state (two
 // supported encodings), the payload field will remain set to
 // "PayloadCodec2"
}

## 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)})]

 ")"

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)})

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)})

/\* 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 ::= EncodeAttribute |

 VariantAttribute |

 DisplayAttribute |

 ExtensionAttribute |

 OptionalAttribute

 [AttribKeyword](#TAttribKeyword) [[OverrideKeyword](#TOverrideKeyword)] [[AttribQualifier](#TAttribQualifier)]

 [FreeText](#TFreeText)

469. EncodeAttribute ::= [EncodeKeyword](#TEncodeKeyword) [( [OverrideKeyword](#TOverrideKeyword) | LocalModifier | OnlyModifier )]

 [[AttribQualifier](#TAttribQualifier)] [FreeText](#TFreeText)

470. VariantAttribute ::= [VariantKeyword](#TVariantKeyword) [( [OverrideKeyword](#TOverrideKeyword) | LocalModifier )]

 [[AttribQualifier](#TAttribQualifier)] [FreeText](#TFreeText) [ "." FreeText ]

471. DisplayAttribute ::= [DisplayKeyword](#TDisplayKeyword) [( [OverrideKeyword](#TOverrideKeyword) | LocalModifier )]

 [[AttribQualifier](#TAttribQualifier)] [FreeText](#TFreeText)

472. ExtensionAttribute ::= [ExtensionKeyword](#TExtensionKeyword) [( [OverrideKeyword](#TOverrideKeyword) | LocalModifier )]

 [[AttribQualifier](#TAttribQualifier)] [FreeText](#TFreeText)

473. OptionalAttribute ::= [OptionalKeyword](#TOptionalKeyword) [( [OverrideKeyword](#TOverrideKeyword) | LocalModifier )]

 [[AttribQualifier](#TAttribQualifier)] [FreeText](#TFreeText)

EncodeKeyword ::= "encode"

VariantKeyword ::= "variant"

DisplayKeyword ::= "display"

ExtensionKeyword ::= "extension"

OverrideKeyword ::= "override"

LocalModifier ::= "@local"

OnlyKeyword ::= "@only"

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"

# C.5 Codec functions

## C.5.1 The encoding function

 **encvalue**(**in template (value)** any\_typeinpar,

 **in** **universal charstring** encoding\_info := "",

 **in** **universal charstring** dynamic\_encoding := "") **return bitstring**

The **encvalue** function encodes a value or template into a bitstring. When the actual parameter that is passed to inpar is a template, it shall resolve to a specific value (the same restrictions apply as for the argument of the **send** statement). The returned bitstring represents the encoded value of inpar, however, the TTCN-3 test system need not make any check on its correctness. The optional encoding\_info parameter is used for passing additional encoding information to the codec and, if it is omitted, no additional information is sent to the codec.

The optional dynamic\_encoding parameter is used for dynamic selection of encode attribute of the inpar value for this single encvalue call. The rules for dynamic selection of the encode attribute are described in 27.9.

In addition to the general error causes in clause 16.1.2, error causes are:

* Encoding fails due to a runtime system problem (i.e. no encoding function exists for the actual type of inpar).

## C.5.2 The decoding function

 **decvalue**(**inout bitstring** encoded\_value**, out** any\_typedecoded\_value,

 **in** **universal charstring** decoding\_info := "",

 **in** **universal charstring** dynamic\_encoding := "") **return integer**

The **decvalue** function decodes a bitstring into a value. The test system shall suppose that the bitstring encoded\_value represents an encoded instance of the actual type of decoded\_value. The optional decoding\_info parameter is used for passing additional decoding information to the codec and, if it is omitted, no additional information is sent to the codec.

The optional dynamic\_encoding parameter is used for dynamic selection of encode attribute of the decoded\_value parameter for this single decvalue call. The rules for dynamic selection of the encode attribute are described in 27.9.

If the decoding was successful, then the used bits are removed from the parameter encoded\_value, the rest is returned (in the parameter encoded\_value), and the decoded value is returned in the parameter decoded\_value. If the decoding was unsuccessful, the actual parameters for encoded\_value and decoded\_value are not changed. The function shall return an integer value to indicate success or failure of the decoding below:

* The return value 0 indicates that decoding was successful.
* The return value 1 indicates an unspecified cause of decoding failure. This value is also returned if the encoded\_value parameter contains an unitialized value.
* The return value 2 indicates that decoding could not be completed as encoded\_value did not contain enough bits.

The restrictions in clause 16.1.2 apply.

## C.5.3 The encoding to universal charstring function

 **encvalue\_unichar**(**in template** (**value**)any\_typeinpar,

 **in** **charstring** string\_serialization := "UTF-8",

 **in** **universal charstring** encoding\_info := "",

 **in** **universal charstring** dynamic\_encoding := "")

 **return universal charstring**

The **encvalue\_unichar** function encodes a value or template into a universal charstring. When the actual parameter that is passed to inpar is a template, it shall resolve to a specific value (the same restrictions apply as for the argument of the **send** statement). The returned universal charstring represents the encoded value of inpar, however, the TTCN-3 test system need not make any check on its correctness. If the optional string\_serialization parameter is omitted, the default value "UTF-8" is used. The optional encoding\_info parameter is used for passing additional encoding information to the codec and, if it is omitted, no additional information is sent to the codec.

The optional dynamic\_encoding parameter is used for dynamic selection of encode attribute of the inpar value for this single encvalue\_unichar call. The rules for dynamic selection of the encode attribute are described in 27.9.

The following values (see ISO/IEC 10646 [2]) are allowed as string\_serialization actual parameters (for the description of the UCS encoding scheme see clause 27.5):

1. "UTF-8"
2. "UTF-16"
3. "UTF-16LE"
4. "UTF-16BE"
5. "UTF-32"
6. "UTF-32LE"
7. "UTF-32BE"

The serialized bitstring shall not include the optional signature (see clause 10 of ISO/IEC 10646 [2], also known as byte order mark).

In case of "UTF-16" and "UTF-32" big-endian ordering shall be used (as described in clauses 10.4 and 10.7 of ISO/IEC 10646 [2]).

The specific semantics of this function are explained by the following TTCN-3 definition:

 **function** encvalue\_unichar(**in** **template**(**value**) any\_type inpar,

 **in** **charstring** enc

 **in** **universal** **charstring** encoding\_info := "",

 **in** **universal charstring** dynamic\_encoding := "")

 **return** **universal** **charstring** {

 **return** **oct2unichar**(**bit2oct**(**encvalue**(inpar, encoding\_info, dynamic\_encoding)), enc);

 }

The encvalue\_unichar function first invokes the encvalue function in order to encode the value passed in the inpar parameter to a bitstring. The bitstring is then converted to an octetstring by the bit2oct function and subsequently to a universal charstring using the oct2unichar function. The string\_serialization parameter defines how the encoded octets (in fact the encoded bitstring received from the codec) contain the characters. The universal charstring value is then returned as the result of the encvalue\_unichar function.

In addition to the general error causes in clause 16.1.2, error causes are:

* Encoding fails due to a runtime system problem (i.e. no encoding function exists for the actual type of inpar).
* The given string encoding is not recognized.

## C.5.4 The decoding from universal charstring function

 **decvalue\_unichar**(**inout universal charstring** encoded\_value**,**

 **out** any\_typedecoded\_value,

 **in charstring** string\_serialization:= "UTF-8",

 **in** **universal** **charstring** decoding\_info := "",

 **in** **universal charstring** dynamic\_encoding := "")

 **return integer**

The **decvalue\_unichar** function decodes (part of) a universal charstring into a value. The test system shall suppose that a prefix of the universal charstring encoded\_value represents an encoded instance of the actual type of decoded\_value. The optional decoding\_info parameter is used for passing additional decoding information to the codec and, if it is omitted, no additional information is sent to the codec.

The optional dynamic\_encoding parameter is used for dynamic selection of encode attribute of the decoded\_value parameter for this single decvalue\_unichar call. The rules for dynamic selection of the encode attribute are described in 27.9.

If the decoding was successful, then the characters used for decoding are removed from the parameter encoded\_value, the rest is returned (in the parameter encoded\_value), and the decoded value is returned in the parameter decoded\_value. If the decoding was unsuccessful, the actual parameters for encoded\_value and decoded\_value are not changed. The function shall return an integer value to indicate success or failure of the decoding below:

* The return value 0 indicates that decoding was successful.
* The return value 1 indicates an unspecified cause of decoding failure. This value is also returned if the encoded\_value parameter contains an unitialized value.
* The return value 2 indicates that decoding could not be completed as encoded\_value did not contain enough bits.

If the optional string\_serialization parameter is omitted, the default value "UTF-8" is used.

The following values (see ISO/IEC 10646 [2]) are allowed as string\_serialization actual parameters (for the description of the UCS encoding scheme see clause 27.5):

1. "UTF-8"
2. "UTF-16"
3. "UTF-16LE"
4. "UTF-16BE"
5. "UTF-32"
6. "UTF-32LE"
7. "UTF-32BE"

The serialized bitstring shall not include the optional signature (see clause 10 of ISO/IEC 10646 [2], also known as byte order mark).

In case of "UTF-16" and "UTF-32" big-endian ordering shall be used (as described in clauses 10.4 and 10.7 of ISO/IEC 10646 [2]).

The semantics of the function can be explained by the following TTCN-3 function:

**function** decvalue\_unichar (**inout** **universal** **charstring** encoded\_value,

 **out** any\_type decoded\_value,

 **in** **charstring** string\_encoding := "UTF-8"",

 **in** **universal** **charstring** decoding\_info := "",

 **in** **universal charstring** dynamic\_encoding := "") **return** **integer** {

 **var** **bitstring** v\_str = **oct2bit**(**unichar2oct**(encoded\_value, string\_encoding));

 **var** **integer** v\_result := **decvalue**(v\_str, decoded\_value, decoding\_info,

dynamic\_encoding);

 **if** (v\_result == 0) { // success

 encoded\_value := **oct2unichar**(**bit2oct**(v\_str), string\_encoding);

 }

 **return** v\_result;

}

The decvalue\_unichar function first converts the universal charstring value passed in the encoded\_value parameter into an octetstring using the unichar2oct function. The string\_encoding parameter controls how the characters are converted into octets (in fact how the bitstring sent to the codec contains the characters). The octetstring is subsequently converted into a bitstring by the oct2bit function. This bitstring is then passed as a parameter to the standard decvalue function that performs the actual decoding. In case of successful decoding, the undecoded part of the message is automatically converted from bitstring to octetstring by the bit2oct function and then to universal charstring using the oct2unichar function. This universal charstring is then assigned to the encoded\_value parameter. The result of decoding is then returned to the TE, finishing the decvalue\_unichar call.

The restrictions in clause 16.1.2 apply.

## C.5.5 Retrieving the type of string encoding

**get\_stringencoding(in octetstring** encoded\_value**) return charstring**

The **get\_stringencoding** function analyses the encoded\_value and returns the UCS encoding scheme according to clause 10 of ISO/IEC 10646 [2] (see also clause 27.5 of the present document). The identified encoding scheme, or the value "<unknown>", if the type of encoding cannot be determined unanimously, shall be returned as a character string.

NOTE: The initial octet sequence (also known as byte order mark, BOM), when present, allows identifying the encoding scheme unanimously. When it is not present, other symptoms may be used to identify the encoding scheme unanimously; for example, only UTF-8 may have odd number of octets **and** bit distribution according to table 2 of clause 9.1 of ISO/IEC 10646 [2].

EXAMPLE:

 **match** ( **get\_stringencoding**('6869C3BA7A'O,**charstring**:"UTF-8") ) // true
 //(the octetstring contains the UTF-8 encoding of the character sequence "hiúz")

## C.5.6 Removing BOMs of UCS encoding schemes

 **remove\_bom**(**in octetstring** encoded\_value) **return octetstring**

The **remove\_bom** function removes the optional FEFF ZERO WIDTH NO-BREAK SPACE sequence that may be present at the beginning of a stream of serialized (encoded) universal character strings to indicate the order of the octets within the encoding form, as defined in clause 10 of ISO/IEC 10646 [2]. If no FEFF ZERO WIDTH NO-BREAK SPACE sequence present in the encoded\_value parameter, the function shall return the value of the parameter without change.

Table C.2: Overview of initial octet sequences used for BOM

|  |  |  |
| --- | --- | --- |
| Coding scheme | initial octet sequence | comments |
| UTF-8 | EF BB BF | signature not required / no effect |
| UTF-16BE | FE FF | no signature meaning |
| UTF-16LE | FF FE | no signature meaning |
| UTF-16 | FE FFFF FE | signature(default FE FF) |
| UTF-32BE | 00 00 FE FF | no signature meaning |
| UTF-32LE | FF FE 00 00 | no signature meaning |
| UTF-32 | 00 00 FE FFFF FE 00 00 | signature(default 00 00 FE FF) |

EXAMPLE:

**remove\_bom**('FEFF0068006900FA007A'O) // returns '0068006900FA007A'O

**remove\_bom**('BC'O) ) // returns 'BC'O

// note that this octetstring doesn't contain valid UCS character

//example use: automatic decoding of encoded character strings:

**oct2unichar**(**remove\_bom**(v\_myEncodedCharacterSequence),

 **get\_stringencoding**(v\_myEncodedCharacterSequence))