ETSI ES 201 873-1 V4.7.1 (2015-06)

Methods for Testing and Specification (MTS);

The Testing and Test Control Notation version 3;

Part 1: TTCN‑3 Core Language

**ETSI Standard**

Reference

RES/MTS-201873-1 T3ed471

Keywords

language, methodology, testing, TTCN-3

***ETSI***

650 Route des Lucioles

F-06921 Sophia Antipolis Cedex - FRANCE

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

Siret N° 348 623 562 00017 - NAF 742 C

Association à but non lucratif enregistrée à la

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

***Important notice***

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

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

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

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

***Copyright Notification***

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

© European Telecommunications Standards Institute 2015.

All rights reserved.

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

### 6.2.4 Enumerated type and values

TTCN‑3 supports **enumerated** types. Enumerated types are used to model types that take only a distinct named set of values. Such distinct values are called enumerated values. Each enumerated value shall have an identifier. Operations on enumerated types shall only use these identifiers and are restricted to assignment, equivalence and ordering operators. The identifiers of enumerated values shall be unique within the enumerated type (but do not have to be globally unique) and are consequently visible in the context of the given type only. The identifiers of enumerated values shall only be reused within other structured type definitions and shall not be used for identifiers of local or global visibility at the same or a lower level of the same branch of the scope hierarchy (see scope hierarchy in clause 5.2).

EXAMPLE 1: Declaration of enumerated types and values

**type** **enumerated** MyFirstEnumType {

Monday, Tuesday, Wednesday, Thursday, Friday

};

**type** **integer** Monday;

// This definition does not clash with the previous one

// as Monday in MyFirstEnumType is of local scope

**type** **enumerated** MySecondEnumType {

Saturday, Sunday, Monday

};

// This definition is legal as it reuses the Monday identifier within

// a different enumerated type

**type** **record** MyRecordType {

**integer** Monday

};

// This definition is legal as it reuses the Monday identifier within

// a distinct structured type as identifier of a given field of this type

**type** **record** MyNewRecordType {

MyFirstEnumType firstField,

**integer** secondField

};

**var** MyNewRecordType newRecordValue := { Monday, 0 }

// MyFirstEnumType is implicitly referenced via the firstField element of MyNewRecordType

Each enumerated value may optionally have a user-assigned integer value or non-empty list of integer literal values or ranges of integer literal values, which is defined after the name of the enumerated value in parenthesis. Each user-assigned integer number shall be distinct within a single **enumerated** type, all ranges of all the values lists shall be disjoint and shall not include any of the used single integer values. For each enumerated value without an assigned integer value or list, the system successively associates an integer number in the textual order of the enumerated values, starting at the left-hand side, beginning with zero, by step 1 and skipping any number occupied by any of the enumerated values with a manually assigned value or value list. These values are only used by the system to allow the use of relational operators. Enumerated names with an associated value list can only be used as values together with a specific integer value (which must be one of the associated list) in parenthesis after the name. They can be used as a template of the enumerated type by adding a list of integer template in parenthesis after the name. For enumerated values with no value assigned or with an explicit integer value assigned, the user shall not directly use associated integer values but can access them and convert integer values into enumerated values by using the predefined functions **enum2int** and **int2enum** (see clauses 16.1.2, C.1.30 and C.1.4).

NOTE 1: The integer value also may be used by the system to encode/decode enumerated values. This, however is outside the scope of the present document (with the exception that TTCN‑3 allows the association of encoding attributes to TTCN‑3 items).

For any instantiation or value reference of an **enumerated** type, the given type shall be implicitly or explicitly referenced.

NOTE 2: If the enumerated type is an element of a user defined structured type, the enumerated type is implicitly referenced via the given element (i.e. by the identifier of the element or the position of the value in a value list notation) at value assignment, instantiation, etc.

EXAMPLE 2: Using enumerated types (see also example 4 of clause 8.2.3.1)

// Valid instantiations of MyFirstEnumType and MySecondEnumType would be

**var** MyFirstEnumType Today := Tuesday;

**var** MySecondEnumType Tomorrow := Monday;

// The following statements however cause an error as the two variables are instances

// of different enumeration types

Today := Tomorrow;

Today == Tomorrow;

// The following operation is correct

**if** (Today == Monday ) {...}

// the type of variable Today identifies the type context of MyFirstEnumType for the

// equality operator

// But the following causes an error

**if** ( Tuesday == Wednesday ) {...}

// there is no TTCN-3 type(d) object to establish the type context for the equality operator

// Please note that the values Tuesday and Wednesday are defined within the type

// MyFirstEnumType only, but this is not sufficient to establish the type context

type enumerated MyThirdEnumType {

Blue(0),

Yellow(1),

Green(3),

Other(2, 4..255)

}

Var MyThirdEnumType color := Other(5);

if (color == Other(4)) { // is false

}

if (color > Other(4)) { // is true

}

if (match(color, Other(?))) { // is true

}

if (match(color, Other(6..10))) { // is false

}

When a TTCN-3 module parameter, formal parameter, constant, variable, non-parameterized template or parameterized template with all formal parameters having default values of an imported enumerated type is defined, the name of that definition shall not be the same as any of the enumerated values of that type.

### 7.1.3 Relational operators

The predefined relational operators are equality (==), less than (<), greater than (>), non‑equality to (!=), greater than or equal to (>=) and less than or equal to (<=). The result type of all these operations is **boolean**.

The relational operators less than (<), greater than (>), greater than or equal to (>=), and less than or equal to (<=) shall have only operands of type **integer** (including derivations of **integer**), **float** (including derivations of **float**), or instances of the same **enumerated** type. It is not allowed to compare instances of different root types.

The **address** type is allowed for the equality (==) and non-equality (!=) operators, independent of its actual type, but when its actual type differs from the types specified above, it can be compared to the literal special value **null** only.

Operands of equality (==) and non-equality (!=) shall be completely initialized values or field references of type compatible root types and the values or field references being compared shall obey the following rules. This implies that instances of types not mentioned below shall not be operands of equality and non-equality.

* Two field references are equal if the referenced fields are both **optional** fields and both fields are set to **omit** or if both referenced fields (regardless if they are optional or not) are initialized with values and these values are equal. A field reference is equal to a value if the referenced field is initialized with a value and both values are equal.
* Two integer values are equal if and only if they contain the same value. Otherwise, normal mathematical ordering is applied.
* Two enumerated values are equal if and only if they are associated with the same integer value. Otherwise, they are ordered using the mathematical order on the associated integer values.
* Two floating-point numbers are equal if and only if they contain the same value. The values minus zero and plus zero are two distinct values (e.g. they are encoded differently in some standardized languages) and minus zero is less than plus zero, which represents zero. Otherwise, normal mathematical ordering is applied. The special values **‑infinity,** **infinity** and **not\_a\_number** are equal to themselves only. The special value **‑infinity** is less than any other float value. The special value **infinity** is greater than any numerical float values and **-infinity**. The special value **not\_a\_number** is greater than any other float value (including **infinity)**.
* Two charstring or two universal charstring values are equal if and only if they have equal lengths and the characters at all positions are the same.
* For values of bitstring, hexstring or octetstring types, the same equality rule applies as for charstring values with the exception, that fractions which shall equal at all positions are bits, hexadecimal digits or pairs of hexadecimal digits accordingly.
* Two record values, or set values are equal respectively if and only if they are mutually compatible with the type of the other operand (see clause 6.3), the actual values of all present fields are equal to their corresponding fields and all fields corresponding to omitted fields are also omitted in the peer value.
* Two record of values, set of values or array values, respectively, are equal if and only if they are mutually compatible with the type of the other operand (see clause 6.3), they both have the same length, and and each element of one value is equal to the corresponding element of the other value. Record of values and array values may also be compared, in which case the corresponding record of type of the array is being considered.
* Values of the same union type, and values of different union types in which at least one of the alternatives is compatible with the other type (see clause 6.3.2.4) can be compared (independent if a compatible alternative is the selected one or not). Two values of union types are equal if and only if in both values the name of the selected alternative is identical, they are compatible with the type of the other value, and the actual values of the chosen fields are equal.
* Values of the same or any two anytype types can be compared. For anytype values the same rule apply as to union values, with the addition that names of types defined with the same name in different modules do not denote the same name of the selected alternatives.
* Two default or two component values are equal if and only if they contain the same value (i.e. they designate the same default or test component, independent of the actual state of the denoted object).
* It is also possible to use compound expressions (field assignment or value list notation) directly as operands of comparison operations of structured types. If there is a compound expression on both sides of the comparison operator, they shall both be value list notation expressions where the elements shall be of the same root type and they shall be compared like record of values with elements of that root type. If only one operand of the comparison operation is a compound expression it shall be compatible with the root type of the other operand and they shall be compared like values of that root type.

EXAMPLE:

// Given

**type** **set** S1 {

**integer** a1 **optional**,

**integer** a2 **optional**,

**integer** a3 **optional**

};

**type** **set** S2 {

**integer** b1 **optional**,

**integer** b2 **optional**,

**integer** b3 **optional**

};

**type** **set** S3 {

**integer** c1 **optional**,

**integer** c2 **optional**,

};

**type** **set** **of** **integer** SI;

**type** **union** U1 {

**integer** d1,

**integer** d2,

};

**type** **union** U2 {

**integer** e1,

**integer** e2,

};

**type** **union** U3 {

**integer** d1,

**integer** d2,

**boolean** d3

};

// And

**const** S1 s1 := { a1 := 0, a2 := **omit**, a3 := 2 };

// Notice that the order of defining values of the fields does not matter

**const** S2 s2a := { b1 := 0, b3 := 2, b2 := **omit** };

**const** S2 s2b := { b2 := 0, b3 := 2, b1 := **omit** };

**const** S3 s3 := { c1 := 0, c2 :=2 };

**var** SI v\_si:= { 0, -, 2 };

**const** SI si := { 0, 2 };

**const** U1 u1 := { d1:= 0 };

**const** U2 u2 := { e1:= 0 };

**const** U3 u3; := { d1:= 0 };

// Then

s1 == s2a;

// returns **true**

s1 == s2b;

// returns **false**, because neither a1 nor a2 are equal to their counterparts

// (the corresponding element is not omitted)

s1 == s3;

// returns **false**, because the effective value structures of s1 and s3 are not compatible

s1 == v\_si;

// causes test case error as v\_si is not completely initialized  
 // (2nd element is left uninitialized)

s1 == si;

// returns **false**, as the counterpart of the omitted a2 is 2,

// but the counterpart of a3 is undefined

s3 == si;

// returns **true**

u1 == u2;

// causes error as U1 and U2 have no common subset of alternatives

u1 == u3;

// returns **true**, as alternatives with the same names are chosen and

// the actual values in the selected alternatives are equal

{ 0, omit, 2 } == s1;

// returns true

s2a == { b1 := 0, b2:= omit, b3 := 2 };

// returns true

{ s1, s2b } == { s2a, s1 };

// returns false because s2b != s1

{ s1, s2b, s2a } == { s1 };

// returns false because of different length

s1.a1 == s2a.b1;

// returns true, both fields are initialized with values and the values are equal

s1.a2 == s2a.b2;

// returns true, both fields are omit

s1.a1 == s2a.b2;

// returns false, value vs. omit

s1.a1 == omit;

// error, omit is neither a value nor a field reference

s1.a2 == 3;

// false, omit vs. value

### 8.2.3 Importing from modules

It is possible to re-use visible definitions specified in different modules using the **import** statement. Every definition in a TTCN‑3 module has an associated visibility, which is by default **public** (see clause 8.2.5).

NOTE: Groups are **public** only. Importing a group means that only the visible elements of the group are being imported.

#### 8.2.3.1 General format of import

An import statement can be used anywhere in the module definitions part.

***Syntactical Structure***

[ *Visibility* ] **import** **from** *ModuleId*

(

( **all** [ **except** "{" *ExceptSpec* "}" ] )

|

( "{" *ImportSpec* "}" )

)

[ ";" ]

***Semantic Description***

TTCN‑3 supports the import of the following definitions: module parameters, user defined types, signatures, constants, data templates, signature templates, functions, external functions, altsteps and test cases. Each definition has a *name* (defines the identifier of the definition, e.g. a function name), a *specification* (e.g. a type specification or a signature of a function) and in the case of functions, altsteps and test cases an associated *behaviour description*. In addition, import statements of one module can be explicitly imported by another module (see clause 8.2.3.7). Only definitions or import statements visible from the importing module can be imported (see clause 8.2.5).

In contrast to module definitions, which are by default public, import statements are by default private.

EXAMPLE 1a:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Name | Specification | Behaviour description |
| **function** | MyFunction | (**inout** MyType1 MyPar) **return** MyType2 **runs on** MyCompType | {  const MyType3 MyConst := …;  : // further behaviour  } |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Specification | Name | Specification |
| **type** | **record** | MyRecordType | {  MyType4 field1,  integer field2  } |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Specification | Name | Specification |
| **template** | MyType5 | MyTemplate | := {  field1 := 1,  field2 := MyConst, // MyConst is a module constant  field3 := ModulePar // ModulePar is module parameter  } |

Behaviour descriptions have no effect on the import mechanism, because their internals are considered to be invisible to the importer when the corresponding functions, altsteps or test cases are imported. Thus, they are not considered in the following descriptions.

The specification part of an importable definition contains *local definitions* (e.g. field names of structured type definitions or values of enumerated types) and *referenced definitions* (e.g. references to type definitions, templates, constants or module parameters). For the examples above, this means:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Name | Local definitions | Referenced definitions |
| **function** | MyFunction | MyPar | MyType1, MyType2, MyCompType |
| **type** | MyRecordType | field1, field2 | MyType4, integer |
| **template** | MyTemplate |  | MyType5, field1, field2, field3, MyConst, ModulePar |

NOTE 1: The local definitions column refers to identifiers only that are newly defined in the importable definition. Values assigned to individual fields of importable definitions, e.g. in template definitions, may also be considered as local definitions, but they are not important for the explanation of the import mechanism.

NOTE 2: The referenced definitions field1, field2 and field3 of template MyTemplate are the field names of MyType5, i.e. they are referenced via MyType5.

Referenced definitions are also importable definitions, i.e. the source of a referenced definition can again be structured into a name and a specification part and the specification part also contains local and referenced definitions. In other words, an importable definition may be built up recursively from other importable definitions.

The TTCN‑3 import mechanism is related to the local and referenced definitions used in the specification part of the importable definitions. Table 8 specifies the possible local and referenced definitions of importable definitions.

Table 8: Possible local and referenced definitions of importable definitions

|  |  |  |
| --- | --- | --- |
| Importable Definition | Possible Local Definitions | Possible Referenced Definitions |
| Module parameter |  | Module parameter type |
| User-defined type (for all) |  |  |
| * enumerated type | Concrete values |  |
| * structured type | Field names, nested type definitions | Field types |
| * port type |  | Message types, signatures |
| * component type | Constant names, variable names, timer names and port names | Constant types, variable types, port types |
| Signature | Parameter names | Parameter types, return type, types of exceptions |
| Constant |  | Constant type |
| Data Template | Parameter names | Template type, parameter types, constants, module parameters, functions |
| Signature template |  | Signature definition, constants, module parameters functions |
| Function | Parameter names | Parameter types, return type, component type (**runs on** clause) |
| External function | Parameter names | Parameter types, return type |
| Altstep | Parameter names | Parameter types, component type (**runs on** clause) |
| Test case | Parameter names | Parameter types, component types (**runs on**- and **system** clause) |
| NOTE 1: For the import of import statements see clause 8.2.3.7.  NOTE 2: For the import of groups see clause 8.2.3.3. | | |

The TTCN‑3 import mechanism distinguishes between the *identifier of a referenced definition* and the *information necessary for the usage of a referenced definition* within the imported definition. For the usage, the identifier of a referenced definition is not required and therefore not imported automatically.

EXAMPLE 1b: Differentiation between *information necessary for the usage* and the identifier.

**module** A {

**type** **record** MyRec1 {

**integer** field1,

**charstring** field2

}

}

**module** B {

**import** **from** A **all**;

**type** **record** MyRec2 {

MyRec1 myField1,

// "myField1" is the local definition, "MyRec1" is a referenced definition;

// the *name* "MyRec1" shall be imported in this case as is directly referenced

**boolean** myField2

}

}

**module** C {

**import** **from** B **all**;

**const** MyRec2 t\_MyRec2 := {

myField1 := { field1 := 5, field2 := "A" },

// to define myField1 of MyRec2 the name "MyRec1" is not needed, the

// *information necessary for the usage* is its type information,

// i.e. names and types of its fields field1 and field2

// which is embeddded in the imported definition of MyRec2

myField2 := **true**

}

}

If an imported definition has attributes (defined by means of a **with** statement) then the attributes shall also be imported. The mechanism to change attributes of imported definitions is explained in clause 27.1.3.

NOTE 3: If the module has global attributes they are associated to definitions without these attributes.

The use of **import** on single definitions, groups of definitions, definitions of the same kind, etc. may lead to situations where the *same definition is referred to more than once*. Such cases shall be resolved by the system and definitions shall be imported only once.

NOTE 4: The mechanisms to resolve such ambiguities, e.g. overwriting and sending warnings to the user, are outside the scope of the present document and should be provided by TTCN‑3 tools.

All **import** statements and definitions within import statements are considered to be treated independently one after the other in the order of their appearance.

All TTCN‑3 modules shall have their own name space in which all definitions shall be uniquely identified. *Name clashes* may occur due to import, e.g. import from different modules. Name clashes shall be resolved using qualified name(s) for the imported definition(s), i.e. prefixing the imported definition (which causes the name clash) by the identifier of the module in which it has been defined; the prefix and the identifier shall be separated by a dot ("."). If the type of the component referenced in a connection operation is known (either when the component reference is a variable or value returned from a function or the type is defined the runs on, mtc or system clause of the calling function), the referenced port declaration shall be present in this component type.

There is one exception to this rule: when **in the context** of an enumerated type (see clause 6.2.4), an enumerated value is clashing with the name of a definition in the importing module, the enumerated value shall take precedence and the definition in the importing module shall be referenced by using its qualified name (see example 4 below in this clause).

In cases where there are no ambiguities the prefixing need not (but may) be present when the imported definitions are used. When the definition is referenced in the same module where it is defined, the module identifier of the module (the current module) also may be used for prefixing the identifier of the definition. For the latter case, prefixing shall only be used for definitions with global visibility for the module.

***Restrictions***

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

a) An import statement shall only be used in the module definitions part and not be used within a control part, function definition, and alike.

b) Only top-level visible definitions of a module may be imported. Definitions which are top-level but invisible to the importing module or which occur at a lower scope (e.g. local constants defined in a function) shall not be imported.

c) A definition is imported together with its name and all local definitions.

NOTE 5: A local definition, e.g. a field name of a user-defined record type or an enumerated value, has only meaning in the context of the definitions in which it is defined, e.g. a field name of a record type can only be used to access a field of the record type and not outside this context.

In particular, importing an enumerated type does not impose the restriction given in clause 6.2.4 on global names defined in the importing module.

d) A definition is imported together with all information of referenced definitions that are necessary for the usage of the imported definition, independent of the visibility of the referenced definitions (see clause 8.2.5).

NOTE 6: If module C imports a definition from module B that uses a type reference defined in module A, the corresponding information necessary for the usage of that type is automatically imported into module C (see example 5 below in this clause). Identifiers of referenced definitions are not automatically imported.

In particular, if module C imports global value or template definitions (e.g. constants, module parameters, templates) or local definitions (e.g. formal parameters of templates, functions, etc., or constants and variables of component types) of an enumerated type from module B, the enumerated values of this type (i.e. the identifiers) are implicitly and automatically imported to module C. That is, the enumerated values are known when an enumerated value or template is used in module C (e.g. when an actual parameter is passed or a value is assigned to a component variable). Note that this implicit importing does not impose the restriction given in clause 6.2.4 on global names defined in module C.

e) If the referenced definitions are wished to be used in the importing module, they shall be explicitly imported either directly from its source module or indirectly by importing the import statements of a module importing it (see clause 8.2.3.7).

f) When importing a function, altstep or test case the corresponding behaviour specifications and all definitions used inside the behaviour specifications remain invisible for the importing module.

g) The language specification (see clause 8.1) of the import statement shall not override the language specification of the importing module.

h) The language specification of the import statement shall be identical to the language specification of the source module from which definitions are imported (see clause 8.2.3.8) provided a language specification is defined in the source module. If not, the language specification in the import statement is taken as the language specification of the source module. If the source module uses however language concepts not being part of that language specification, this causes an error for the import statement.

***Examples***

EXAMPLE 1: Selected import examples

**module** MyModuleA

{ :

// Scope of the imported definitions is global to MyModuleA

**import** **from** MyModuleB **all**; // import of all definitions from MyModuleB

**import from** MyModuleC { // import of selected definitions from MyModuleC

**type** MyType1, MyType2; // import of types MyType1 and MyType2

**template** **all** // import of all templates

}

:

**function** MyBehaviourC()

{

// import cannot be used here

:

}

**:**

**control**

{

// import cannot be used here

:

}

}

EXAMPLE 2: Use of imported definitions and visibility of definitions referenced by them

**module** ModuleONE {

**modulepar** **integer** ModPar1 := …;

**type** **record** RecordType\_T1 {

**integer** Field1\_T1,

**:**

}

**type** **record** RecordType\_T2 {

RecordType\_T1 Field1\_T2,

:

}

**const** **integer** MyConst := …;

**template** RecordType\_T2 Template\_T2 (RecordType\_T1 TempPar\_T2):= { // parameterized template

Field1\_T2 := …,

:

}

} // end module ModuleONE

**module** ModuleTWO {

**import** **from** ModuleONE {

**template** Template\_T2

}

// Only the names Template\_T2 and TempPar\_T2 will be visible in ModuleTWO. Please note, that

// the identifier TempPar\_T2 can only be used when modifying Template\_T2. All information

// necessary for the usage of Template\_T2, e.g. for type checking purposes, are imported

// for the referenced definitions RecordType\_T1, Field1\_T2, etc., but their identifiers are

// not visible in ModuleTWO.

// This means, e.g. it is not possible to use the constant MyConst or to declare a

// variable of type RecordType\_T1 or RecordType\_T2 in ModuleTWO without explicitly importing  
 // these types.

**import** **from** ModuleONE {

**modulepar** ModPar2

}

// The module parameter ModPar2 of ModuleONE is imported from ModuleONE and

// can be used like an integer constant

} // end module ModuleTWO

**module** ModuleTHREE {

**import** **from** ModuleONE **all**; // imports all definitions from ModuleONE

**type port** MyPortType **message** {

**inout** RecordType\_T2 // Reference to a type defined in ModuleONE

}

**type component** MyCompType {

**var integer** MyComponentVar := ModPar2;

// Reference to a module parameter of ModuleONE

**:**

}

**function** MyFunction () **return** **integer** {

**return** MyConst // Reference to a module constant of ModuleONE

}

**testcase** MyTestCase (**out** RecordType\_T2 MyPar) **runs** **on** MyCompType {

:

MyPort.**send**(Template\_T2); // Sending a template defined in ModuleONE

:

}

} // end ModuleTHREE

**module** ModuleFOUR {

**import from** ModuleTHREE {

**testcase** MyTestCase

}

// Only the name MyTestCase will be visible and usable in ModuleFOUR.

// Type information for RecordType\_T2 is imported via ModuleTHREE from ModuleONE and

// Type information for MyCompType is imported from ModuleTHREE. All definitions

// used in the behaviour part of MyTestCase remain hidden for the user of ModuleFOUR.

} // end ModuleFOUR

EXAMPLE 3: Handling of name clashes

**module** MyModuleA {

:

**type** **bitstring** MyTypeA;

**import from** SomeModuleC {

**type** MyTypeA, // Where MyTypeA is of type character string

MyTypeB // Where MyTypeB is of type character string

}

**:**

**control** {

:

**var** SomeModuleC.MyTypeA MyVar1 := "Test String"; // Prefix shall be used

**var** MyTypeA MyVar2 := '10110011'B; // This is the original MyTypeA

:

**var** MyTypeB MyVar3 := "Test String"; // Prefix need not be used …

**var** SomeModuleC.MyTypeB MyVar3 := "Test String"; // … but it can be if wished

:

}

}

NOTE 7: Definitions with the same name defined in different modules are always assumed to be different, even if the actual definitions in the different modules are identical. For example, importing a type that is already defined locally, even with the same name, would lead to two different types being available in the module.

EXAMPLE 4: Name clash between enumerated values and global definitions

**module** A {

**type** **enumerated** MyEnumType {enumX, enumY}

**type** **enumerated** MyEnumType2 { enumY, enumZ}

}

**module** B {

**import** **from** A **all**;

**const** MyEnumType enumY := enumX; // this is not allowed as enumerated values restrict

// global names (see clause 6.2.4)

**const** MyEnumType2 enumX := enumX;// this is likewise not allowed

const MyEnumType enumZ := enumX; // allowed as MyEnumType does not contain enumZ

}

Module C {

import from A all;

import from B all;

**const** **integer** enumZ := 0;

const integer enumY := 1;

const MyEnumType2 enumX := enumY;

**modulepar** MyEnumType px\_MyModulePar1 := enumY

// the default value of the module parameter will be the value enumY, as the type of

// px\_MyModulePar1 creates the context of MyEnumType and in this context enumerated values

// take precedence over global definition names; note that for the same context reason there

// is no name clash between the enumerated values defined in MyEnumType and in MyEnumType2

**modulepar** MyEnumType px\_MyModulePar2 := B.enumZ

// the default value of the module parameter will be the value enumX, as the prefix

// identifies the constant definition enumZ unambiguously, which has the value enumX

**modulepar** **integer** px\_IntegerPar := enumZ;

// the default value of the module parameter will be 0 as this assignment is not in the

// context of an enumerated type, hence no name clash occurs

**modulepar** MyEnumType px\_MyModulePar3 := C.enumX

// causes an error as px\_MyModulePar3 and the constant enumX has different types

}

EXAMPLE 5: Importing local definitions transitively

**module** A {

**type enumerated** MyEnum\_Type { enumX, enumY, enumZ}

**type record** MyRec { **integer** a, **integer** b }

**type component** MyComp { **var** MyRec v\_Rec := { a := 5 } }

}

**module** B {

**import** **from** A **all**;

**modulepar** MyEnum\_Type px\_MyModulePar := enumY;

**type component** MyCompUser **extends** MyComp {}

}

**module** C {

**import from** B **all**;

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

**if** (px\_MyModulePar == enumY) {

// the enumerated value enumY is know in C without explicitly importing it from A

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

}

**if** (v\_Rec.a == 5) {

v\_Rec.b := v\_Rec.a;

// Both the variable name v\_Rec and the record field names are known in C without

// explicitly importing them from A

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

}

}

}

## 15.7 Template matching mechanisms

Generally, matching mechanisms are used to replace values of single template fields or to replace even the entire contents of a template. Matching mechanisms may also be used in-line (see clause 15.4)*.*

Matching mechanisms are arranged in four groups:

* specific values;
* special symbols that can be used *instead* of values;
* special symbols that can be used *inside* values;
* special symbols which describe *attributes* of values.

Some of the mechanisms may be used in combination.

The supported matching mechanisms and their associated symbols (if any) and the scope of their application are shown in table 11. The left-hand column of this table lists all the TTCN‑3 types to which these matching mechanisms apply. A full description of each matching mechanism can be found in annex B.

***Restrictions***

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

1. All other applications of matching mechanisms than the ones allowed in table 11 are forbidden.

Table 11: TTCN‑3 Matching Mechanisms

| **Used with values of** | **Value** | **Instead of values** | | | | | | | | | | | **Inside values** | | | **Attributes** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | S  p  e  c  i  f  i  c  V  a  l  u  e | | O  m  i  t | C  o  m  p  l  e  m  e  n  t  e  d  L  i  s  t | T e m p l a  t e  L  i  s  t | A  n  y  V  a  l  u  e  (?) | A  n  y  V  a  l  u  e  O  r  N  o  n  e  (\*) | R  a  n  g  e | S  u  p  e  r  s  e  t | S  u  b  s  e  t | P  a  t  t  e  r  n | M  a  t  c  h  d  e  c  o  d  e  d  c  o  n  t  e  n  t | A  n  y  E  l  e  m  e  n  t  (?) | A  n  y  E  l  e  m  e  n  t  s  O  r  N  o  n  e  (\*) | P  e  r  m  u  t  a  t  i  o  n | L  e  n  g  t  h  R  e  s  t  r  i  c  t  i  o  n | I  f  P  r  e  s  e  n  t |
| **boolean** | Yes | | Yes**1** | Yes | Yes | Yes | Yes**1** |  |  |  |  |  |  |  |  |  | Yes**1** |
| **integer** | Yes | | Yes**1** | Yes | Yes | Yes | Yes**1** | Yes |  |  |  |  |  |  |  |  | Yes**1** |
| **float** | Yes | | Yes**1** | Yes | Yes | Yes | Yes**1** | Yes |  |  |  |  |  |  |  |  | Yes**1** |
| **bitstring** | Yes | | Yes**1** | Yes | Yes | Yes | Yes**1** |  |  |  |  | Yes | Yes | Yes |  | Yes | Yes**1** |
| **octetstring** | Yes | | Yes**1** | Yes | Yes | Yes | Yes**1** |  |  |  |  | Yes | Yes | Yes |  | Yes | Yes**1** |
| **hexstring** | Yes | | Yes**1** | Yes | Yes | Yes | Yes**1** |  |  |  |  | Yes | Yes | Yes |  | Yes | Yes**1** |
| **character strings** | Yes | | Yes**1** | Yes | Yes | Yes | Yes**1** | Yes |  |  | Yes | Yes | Yes2 | Yes2 |  | Yes | Yes**1** |
| **record** | Yes | | Yes**1** | Yes | Yes | Yes | Yes**1** |  |  |  |  |  |  |  |  |  | Yes**1** |
| **record of** | Yes | | Yes**1** | Yes | Yes | Yes | Yes**1** |  |  |  |  |  | Yes | Yes | Yes | Yes | Yes**1** |
| **array** | Yes | | Yes**1** | Yes | Yes | Yes | Yes**1** |  |  |  |  |  | Yes | Yes | Yes | Yes | Yes**1** |
| **set** | Yes | | Yes**1** | Yes | Yes | Yes | Yes**1** |  |  |  |  |  |  |  |  |  | Yes**1** |
| **set of** | Yes | | Yes**1** | Yes | Yes | Yes | Yes |  | Yes | Yes |  |  | Yes | Yes |  | Yes | Yes**1** |
| **enumerated** | Yes | | Yes**1** | Yes | Yes | Yes | Yes**1** |  |  |  |  |  |  |  |  |  | Yes**1** |
| **union** | Yes | | Yes**1** | Yes | Yes | Yes | Yes**1** |  |  |  |  |  |  |  |  |  | Yes**1** |
| **anytype** | Yes | | Yes**1** | Yes | Yes | Yes | Yes**1** |  |  |  |  |  |  |  |  |  | Yes**1** |
| NOTE 1: Can be assigned to templates of any type as a whole or to optional fields of record and set templates. However when matching, it shall be applied to optional fields of record and set types only (without restriction on the type of that field).  NOTE 2: Have matching mechanism meaning within character patterns only. | | | | | | | | | | | | | | | | | |

### 15.7.1 Specific values

Specific values are the basic matching mechanism of TTCN-3 templates. Specific values in templates are expressions which do not contain any matching mechanisms.

***Syntactical Structure***

*SingleExpression*

***Semantic Description***

The matching mechanism for a specific value is an expression that evaluates to a specific value.

For further details please refer to clause 6 and to annex B.

***Restrictions***

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

1. See the restrictions given in table 11 and in annex B.

***Examples***

MyPCO.**receive**(**charstring**:"abcxyz");

MyPCO.**receive**('AAAA'O);

### 15.7.2 Special symbols that can be used instead of values

These matching mechanisms can be used to characterize a set of values.

***Syntactical Structure***

**omit** |

"(" { (*TemplateInstance* | **all from** *TemplateInstance*)[","] } ")" |

**complement** "(" { (*TemplateInstance* | **all from** *TemplateInstance*)[","] } ")" |

"?" |

" \*" |

"(" ( *ConstantExpression |* **-infinity** ) ".." ( *ConstantExpression |* **infinity** ) ")" |

**superset** "(" { (*TemplateInstance* | **all from** *TemplateInstance*)[","] } ")" |

**subset** "(" { (*TemplateInstance* | **all from** *TemplateInstance*)[","] } ")" |

**pattern [@nocase]** *Cstring*

**decmatch** ["(" *Expression* ]")" ] *InlineTemplate*

*EnumValueIdentifier*"(" *TemplateBody* {"," *TemplateBody*} ")"

***Semantic Description***

The matching mechanisms for special symbols that can be used *instead* of values are:

* **omit:** the optional field, in which it is used, is not present;

NOTE 1: **omit** can be assigned to templates of any type as a whole or to optional fields of record and set types. **omit** can only be used for matching optional fields.

* **(…):** a list of values or templates;
* **complement (…):** complement of a list of values or templates;
* **?:** wildcard for any value;
* **\*:** wildcard for any value or no value at all, i.e. the field is not present;

NOTE 2: **\*** can be assigned to templates of any type as a whole or to optional fields of record and set types. **\*** can only be used for matching optional fields.

* **(***lowerBound* **..** *upperBound***):** a range of integer or float values between and including the lower- and upper bounds;
* **superset:** at least all of the elements listed, i.e. possibly more;
* **subset:** at most the elements listed, i.e. possibly less;
* **pattern:** a charstring or universal charstring that matches this format;
* **decmatch:** used for matching of encoded payload fields.
* **EnumValueIdentifier with list of templates: used for matching of enum values with associated value list**

The matching mechanisms list, complemented list, subset, and superset can use the elements of a template using the **all from** clause.

For further details please refer to annex B.

***Restrictions***

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

1. See the restrictions given in table 11 and in annex B.
2. All templates and values used in the matching mechanisms above (including the referenced ones, e.g. within a pattern) shall be completely initialized.

***Examples***

MyPCO.**receive** (**integer**:**complement**(1, 2, 3));

Annex A (normative):  
BNF and static semantics

# A.1 TTCN‑3 BNF

This annex defines the syntax of TTCN‑3 using extended BNF (henceforth just called BNF).

## A.1.1 Conventions for the syntax description

Table A.1 defines the metanotation used to specify the extended BNF grammar for TTCN‑3.

Table A.: The syntactic metanotation

|  |  |  |
| --- | --- | --- |
| ::= | is defined to be | definition of non-terminal |
| abc xyz | abc followed by xyz | concatenation |
| | | alternative | alternative |
| [abc] | 0 or 1 instances of abc | optional |
| {abc} | 0 or more instances of abc | repetition 1 |
| {abc}+ | 1 or more instances of abc | repetition 2 |
| {abc}#(n, m) | n to m instances of abc | repetition 3 |
| (...) | textual grouping | grouping |
| Abc | the non-terminal symbol abc | non-terminal |
| "abc" | a terminal symbol abc | terminal |

NOTE: The metanotation defined in table A.1 is parsed from left to right. The metanotation operators have the following precedence, from highest (binding tightest) at the top, to lowest (loosest) at the bottom:

- Repetition, Optional

- Grouping

- Concatenation

- Alternative

- Definition

## A.1.2 Statement terminator symbols

In general all TTCN‑3 language constructs (i.e. definitions, declarations, statements and operations) are terminated with a semi-colon (;). The semi-colon is optional if the language construct ends with a right-hand curly brace (}) or the following symbol is a right-hand curly brace (}), i.e. the language construct is the last statement in a statement block.

## A.1.3 Identifiers

TTCN‑3 identifiers are case sensitive and may only contain lowercase letters (a-z) uppercase letters (A-Z) and numeric digits (0-9). Use of the underscore ( \_ ) symbol is also allowed. An identifier shall begin with a letter (i.e. not with a number and not an underscore).

## A.1.4 Comments

Comments written in free text may appear anywhere in a TTCN‑3 specification. Comments may contain any graphical character defined in ISO/IEC 10646 [2]. Block comments shall be opened by the symbol pair /\* and closed by the symbol pair \*/.

EXAMPLE 1:

/\* This is a block comment

spread over two lines \*/

Block comments shall not be nested.

/\* This is not /\* a legal \*/ comment \*/

Line comments shall be opened by the symbol pair // and closed by a <*newline*>.

EXAMPLE 2:

// This is a line comment

// spread over two lines

EXAMPLE 3:

// The following is not legal

**const** // This is MyConst **integer** MyConst := 1;

// A block comment should have been used instead

**const** /\* This is MyConst \*/ **integer** MyConst := 1;

// A line comment like this works as well

**const** // This is MyConst

**integer** MyConst := 1;

## A.1.5 TTCN‑3 terminals

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

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

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

The predefined function identifiers defined in table 14 and described in annex C shall also be treated as reserved words.

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

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

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

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

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

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

|  |  |  |  |
| --- | --- | --- | --- |
| **@decoded**  **@deterministic** | **@fuzzy** | **@index** | **@lazy**  **@nocase** |

### A.1.5.1 Use of whitespaces and newlines

The elements of the TTCN-3 syntax (reserved words, identifiers, terminal symbols and literal values) shall be separated by whitespace or by special terminal symbols listed in table A.2 according to the TTCN-3 syntax.

In representing whitespace, any one or more of the following characters of the C0 set of Recommendation ITU‑T T.50 [4] and of annex A of Recommendation ITU‑T T.50 [4] may be used in any combination:

* HT - HORIZONTAL TABULATION (9)
* LF - LINE FEED (10)
* VT -VERTICAL TABULATION (11)
* FF - FORM FEED (12)
* CR - CARRIAGE RETURN (13)
* SP - SPACE (32)

The characters of the C0 set of Recommendation ITU‑T T.50 [4] and of annex A of Recommendation ITU‑T T.50 [4] below are denoting newline (end of line). A single CR(13) character directly followed by an LF(10) character denote a single end of line (i.e. the sequence CRLFCRLFVT denotes 3 lines):

* LF - LINE FEED (10)
* VT - VERTICAL TABULATION (11)
* FF - FORM FEED (12)
* CR - CARRIAGE RETURN (13)

Any character or character sequence that is a valid newline is also a valid whitespace.

NOTE: It is recommended that for newline only the CR and LF and for whitespace only the HT, LF, CR and SP control characters are used as the VT and FF characters may cause problems with some conventional text editors.

## 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] ")"] InlineTemplate   
DencodedMatchKeyword ::= "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) ":=" [InLineTemplate](#TInLineTemplate)   
  
/\* STATIC SEMANTICS – if a value parameter is used, the inlinein-line template shall evaluate to a value \*/

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

InLineTemplate ::= [([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 ::= [InLineTemplate](#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) "," [InLineTemplate](#TInLineTemplate) ")"

MatchKeyword ::= "match"

ValueofOp ::= [ValueofKeyword](#TValueofKeyword) "(" [InLineTemplate](#TInLineTemplate) ")"

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) |

[InLineTemplate](#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) ")"

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) "(" [InLineTemplate](#TInLineTemplate) ")" [[ToClause](#TToClause)]

SendOpKeyword ::= "send"

ToClause ::= [ToKeyword](#TToKeyword) ([InLineTemplate](#TInLineTemplate) |

[AddressRefList](#TAddressRefList) |

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

)

AddressRefList ::= "(" [InLineTemplate](#TInLineTemplate) {"," [InLineTemplate](#TInLineTemplate)} ")"

ToKeyword ::= "to"

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

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

CallOpKeyword ::= "call"

CallParameters ::= [InLineTemplate](#TInLineTemplate) ["," [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) "(" [InLineTemplate](#TInLineTemplate) [[ReplyValue](#TReplyValue)] ")" [[ToClause](#TToClause)]

ReplyKeyword ::= "reply"

ReplyValue ::= [ValueKeyword](#TValueKeyword) [Expression](#TExpression)

RaiseStatement ::= [ArrayIdentifierRef](#TArrayIdentifierRef) [Dot](#TDot) [PortRaiseOp](#TPortRaiseOp)

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

[[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) ["(" [InLineTemplate](#TInLineTemplate) ")"] [[FromClause](#TFromClause)]

[[PortRedirect](#TPortRedirect)]

ReceiveOpKeyword ::= "receive"

FromClause ::= [FromKeyword](#TFromKeyword) ([InLineTemplate](#TInLineTemplate) |

[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) ["(" [InLineTemplate](#TInLineTemplate) ")"] [[FromClause](#TFromClause)]

[[PortRedirect](#TPortRedirect)]

TriggerOpKeyword ::= "trigger"

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

PortGetCallOp ::= [GetCallOpKeyword](#TGetCallOpKeyword) ["(" [InLineTemplate](#TInLineTemplate) ")"] [[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) ["(" [InLineTemplate](#TInLineTemplate) [[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) [InLineTemplate](#TInLineTemplate)

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) "," [InLineTemplate](#TInLineTemplate) | [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) ["(" {([FreeText](#TFreeText) |

[InLineTemplate](#TInLineTemplate))

[","]}

")"]

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

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)} "'" "B"

Bin ::= "0" | "1"

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

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

"d" | "e" | "f"

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

### 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) [":=" ([InLineTemplate](#TInLineTemplate) | [Minus](#TMinus))]

RestrictedTemplate ::= [OmitKeyword](#TOmitKeyword) | ([TemplateKeyword](#TTemplateKeyword) [TemplateRestriction](#TTemplateRestriction))

TemplateRestriction ::= "(" ([OmitKeyword](#TOmitKeyword) |

[ValueKeyword](#TValueKeyword) |

[PresentKeyword](#TPresentKeyword)

) ")"

### A.1.6.8 Statements

#### A.1.6.8.1 With statement

WithStatement ::= [WithKeyword](#TWithKeyword) [WithAttribList](#TWithAttribList)

WithKeyword ::= "with"

WithAttribList ::= "{" [MultiWithAttrib](#TMultiWithAttrib) "}"

MultiWithAttrib ::= {[SingleWithAttrib](#TSingleWithAttrib) [[SemiColon](#TSemiColon)]}

SingleWithAttrib ::= [AttribKeyword](#TAttribKeyword) [[OverrideKeyword](#TOverrideKeyword)] [[AttribQualifier](#TAttribQualifier)]

[FreeText](#TFreeText)

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

[VariantKeyword](#TVariantKeyword) |

[DisplayKeyword](#TDisplayKeyword) |

[ExtensionKeyword](#TExtensionKeyword) |

[OptionalKeyword](#TOptionalKeyword)

EncodeKeyword ::= "encode"

VariantKeyword ::= "variant"

DisplayKeyword ::= "display"

ExtensionKeyword ::= "extension"

OverrideKeyword ::= "override"

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

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

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

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

[AllRef](#TAllRef)

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

AllRef ::= ([GroupKeyword](#TGroupKeyword) [AllKeyword](#TAllKeyword) [[ExceptKeyword](#TExceptKeyword) "{" [QualifiedIdentifierList](#TQualifiedIdentifierList)

"}"]) | (([TypeDefKeyword](#TTypeDefKeyword) |

[TemplateKeyword](#TTemplateKeyword) |

[ConstKeyword](#TConstKeyword) |

[AltstepKeyword](#TAltstepKeyword) |

[TestcaseKeyword](#TTestcaseKeyword) |

[FunctionKeyword](#TFunctionKeyword) |

[SignatureKeyword](#TSignatureKeyword) |

[ModuleParKeyword](#TModuleParKeyword)

) [AllKeyword](#TAllKeyword) [[ExceptKeyword](#TExceptKeyword)

"{" [IdentifierList](#TIdentifierList)

"}"])

#### A.1.6.8.2 Behaviour statements

BehaviourStatements ::= [TestcaseInstance](#TTestcaseInstance) |

[FunctionInstance](#TFunctionInstance) |

[ReturnStatement](#TReturnStatement) |

[AltConstruct](#TAltConstruct) |

[InterleavedConstruct](#TInterleavedConstruct) |

[LabelStatement](#TLabelStatement) |

[GotoStatement](#TGotoStatement) |

[RepeatStatement](#TRepeatStatement) |

[DeactivateStatement](#TDeactivateStatement) |

[AltstepInstance](#TAltstepInstance) |

[ActivateOp](#TActivateOp) |

[BreakStatement](#TBreakStatement) |

[ContinueStatement](#TContinueStatement)

SetLocalVerdict ::= [SetVerdictKeyword](#TSetVerdictKeyword) "(" [SingleExpression](#TSingleExpression) {"," [LogItem](#TLogItem)}

")"

SetVerdictKeyword ::= "setverdict"

GetLocalVerdict ::= "getverdict"

SUTStatements ::= [ActionKeyword](#TActionKeyword) "(" [ActionText](#TActionText) {[StringOp](#TStringOp) [ActionText](#TActionText)}

")"

ActionKeyword ::= "action"

ActionText ::= [FreeText](#TFreeText) | [Expression](#TExpression)

ReturnStatement ::= [ReturnKeyword](#TReturnKeyword) [[Expression](#TExpression) | [InLineTemplate](#TInLineTemplate)]   
  
/\* STATIC SEMANTICS - Expression 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) ([Expression](#TExpression) | [TemplateBody](#TTemplateBody))   
  
/\* STATIC SEMANTICS - The Expression 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) | [InLineTemplate](#TInLineTemplate)

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)}+ "}"

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

")" | [ElseKeyword](#TElseKeyword)) [StatementBlock](#TStatementBlock)

/\*\* STATIC SEMANTICS InLineTemplate-s shall be Identifier-s if the UnionKeyword is present in the surrounding SelectCaseConstruct (see clause 19.3.2)\*/

CaseKeyword ::= "case"

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

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"

Annex B (normative):  
Matching values

### B.1.2.10 Matching enumerated value with value list

To match an enumerated value with an associated value list in its definition, the enumerated value name shall be referenced followed by a non-empty list of integer templates in parenthesis.

The template matches only those enumerated values of the same name where the associated integer values is matched by at least one of the integer templates.

***Examples***

**type** **enumerated** Days

{

**Christmas(0), Easter(1), Other(2..365)**

}

template integer greater20 := complement(0 .. 20);

template Days := Other(5..6, greater20); // matches Other(5), Other(6) and Other(21) .. Other(365)

template Days := Other(?); // matches Other(2) .. Other(365)

## C.1.4 Integer to enumerated

**int2enum** ( **in integer** inpar**, out** Enumerated\_type outpar)

This function converts an integer value into an enumerated value of a given enumerated type. The integer value shall be provided as in parameter and the result of the conversion shall be stored in an out parameter. The type of the out parameter determines the type into which the in parameter is converted.

The general error causes in clause 16.1.2 apply.

EXAMPLE:

**type** **enumerated** MyFirstEnumType {

Monday, Tuesday, Wednesday, Thursday, Friday, Weekend(6..7)

};

**type** **enumerated** MySecondEnumType {

Saturday(-3), Sunday (0), Monday

};

//within a dynamic language element:

**var** MyFirstEnumType firstEnum **:=** Tuesday;

**var** MySecondEnumType secondEnum **:=** Sunday;

**int2enum**(0, firstEnum) // firstEnum == Monday

**int2enum**(1, secondEnum) // secondEnum == Monday

int2enum(6, firstEnum) // firstEnum == Weekend(6)

## C.1.30 Enumerated to integer

**enum2int**(**in** Enumerated\_type inpar) **return integer**

This function accepts an enumerated value and returns the **integer** value associated to the enumerated value (see also clause 6.2.4). The actual parameter passed to inpar always shall be a typed object (see clause 6.2.4 and the definition "type context" in clause 3.1).

The general error causes in clause 16.1.2 apply.

EXAMPLE:

**type** **enumerated** MyFirstEnumType {

Monday, Tuesday, Wednesday, Thursday, Friday, Weekend(5..6)

};

**type** **enumerated** MySecondEnumType {

Saturday(-3), Sunday (0), Monday

};

//within a dynamic language element:

**var** MyFirstEnumType vl\_FirstEnum **:=** Monday;

**var** MySecondEnumType vl\_SecondEnum **:=** Monday;

**enum2int(**vl\_FirstEnum**)** // returns 0

**enum2int(**vl\_SecondEnum**)** // returns 1

vl\_FirstEnum **:=** Wednesday;

vl\_SecondEnum **:=** Saturday;

**enum2int(**vl\_FirstEnum**)** // returns 2

**enum2int(**vl\_SecondEnum**)** // returns -3

vl\_FirstEnum **:=** Friday;

vl\_SecondEnum **:=** Sunday;

**enum2int(**vl\_FirstEnum**)** // returns 4

**enum2int(**vl\_SecondEnum**)** // returns 0

v1\_FirstEnum := Weekend(6);

enum2int(v1\_FirstEnum) // returns 6