ETSI ES 202 784 V1.7.1 (2020-05)

Methods for Testing and Specification (MTS);
The Testing and Test Control Notation version 3;
TTCN-3 Language Extensions: Advanced Parameterization

Reference
RES/MTS-202784ed171

Keywords
conformance, 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

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (https://ipr.etsi.org/).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

Foreword

This ETSI Standard (ES) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS).

The use of underline (additional text) and strike through (deleted text) highlights the differences between base document and extended documents.

The present document relates to the multi-part standard ETSI ES 201 873 covering the Testing and Test Control Notation version 3, as identified in ETSI ES 201 873-1 [1].

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

1 Scope

The present document defines the Advanced Parameterization package of TTCN-3. TTCN-3 can be used for the specification of all types of reactive system tests over a variety of communication ports. Typical areas of application are protocol testing (including mobile and Internet protocols), service testing (including supplementary services), module testing, testing of CORBA based platforms, APIs, etc. TTCN-3 is not restricted to conformance testing and can be used for many other kinds of testing including interoperability, robustness, regression, system and integration testing. The specification of test suites for physical layer protocols is outside the scope of the present document.

TTCN-3 packages are intended to define additional TTCN-3 concepts, which are not mandatory as concepts in the TTCN-3 core language, but which are optional as part of a package which is suited for dedicated applications and/or usages of TTCN-3.

This package defines:

- Value parameters of types.
- Type parameterization.

While the design of TTCN-3 package has taken into account the consistency of a combined usage of the core language with a number of packages, the concrete usages of and guidelines for this package in combination with other packages is outside the scope of the present document.

2 References

2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found at https://docbox.etsi.org/Reference.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are necessary for the application of the present document.

[1] ETSI ES 201 873-1: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language".

[2] ETSI ES 201 873-4: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics".

[3] ETSI ES 201 873-5: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI)".

[4] ETSI ES 201 873-6: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI)".

[5] ETSI ES 201 873-7: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 7: Using ASN.1 with TTCN-3".

[6] ETSI ES 201 873-10: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 10: TTCN-3 Documentation Comment Specification".

[7] ISO/IEC 9646-1: "Information technology - Open Systems Interconnection - Conformance testing methodology and framework; Part 1: General concepts".

[8] ETSI ES 201 873-8: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 8: The IDL to TTCN-3 Mapping".

[9] ETSI ES 201 873-9: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 9: Using XML schema with TTCN-3".

[10] Recommendation ITU-T X.683: "Information technology - Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications".

[11] ETSI ES 202 789: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Extended TRI".

[12] ETSI ES 203 790: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Object-Oriented Features".

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

Not applicable.

3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the terms given in ETSI ES 201 873-1 [1], ETSI ES 201 873-4 [2], ETSI ES 201 873-5 [3], ETSI ES 201 873-6 [4], ETSI ES 201 873-7 [5], ETSI ES 201 873-10 [6], ISO/IEC 9646-1 [7] and the following apply:

subtype compatibility: type "A" has a subtype compatibility to another type "B" if all of the values of type A are type compatible with type "B"

NOTE: All subtypes defined via subtype definition are subtype compatible with their supertype.

type parameterization: ability to pass a type as an actual parameter into a parameterized object via a type parameter

NOTE: This actual type parameter is added to the specification of that object and may complete it.

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the abbreviations given in ETSI ES 201 873-1 [1], ETSI ES 201 873-4 [2], ETSI ES 201 873-5 [3], ETSI ES 201 873-6 [4], ETSI ES 201 873-7 [5], ETSI ES 201 873-10 [6] and ISO/IEC 9646-1 [7] apply.

4 Package conformance and compatibility

The package presented in the present document is identified by the package tag:

- `"TTCN-3:2014 Advanced Parameterization"` - to be used with modules complying with the present document.

NOTE: This version of the package only extends the previous versions, identified with the package tag `"TTCN-3:2009 Advanced Parameterization"`, with the option of parameterizing objects - in addition to types - also with signatures. For this reason, modules not containing a formal or actual parameter of the kind signature are compatible with both versions.

For an implementation claiming to conform to this package version, all features specified in the present document shall be implemented consistently with the requirements given in the present document, in ETSI ES 201 873-1 [1] and in ETSI ES 201 873-4 [2].

The package presented in the present document is compatible to:

ETSI ES 201 873-1 [1], version 4.5.1;

ETSI ES 201 873-4 [2], version 4.5.1;

ETSI ES 201 873-5 [3], version 4.5.1;

ETSI ES 201 873-6 [4], version 4.5.1;

ETSI ES 201 873-7 [5], version 4.5.1;

ETSI ES 201 873-8 [8], version 4.5.1;

ETSI ES 201 873-9 [9], version 4.5.1;

ETSI ES 201 873-10 [6], version 4.5.1.

If later versions of those parts are available and should be used instead, the compatibility to the package presented in the present document has to be checked individually.

5 Package concepts for the core language

5.1 Extension to ETSI ES 201 873-1, clause 4 (Introduction)

The present package adds the following essential characteristic to TTCN-3:

- type parameterization.

5.2 Extension to ETSI ES 201 873-1, clause 5 (Basic language elements)

Clause 5.2.1 Scope of formal parameters

Add the following text:

Additionally, formal type parameters can be used as types of formal value parameters, return values, **`runs on`** and **`system`** clauses, where applicable.

Clause 5.4 Parameterization

Additionally, TTCN-3 supports type and signature parameterization.

Replace table 2 "Overview of parameterizable TTCN-3 objects" with the following table.

Table 2: Overview of parameterizable TTCN-3 objects

Keyword	Allowed kind of Parameterization	Allowed form of non-type Parameterization	Allowed types in formal non-type parameter lists
module	Value parameterization	Static at start of run-time	all basic types, all user-defined types and **address** type.
type (notes 1 and 2)	Value parameterization, type parameterization, signature parameterization	Static at compile-time	all basic types, all user-defined types and **address** type.
template	Value and template parameterization, type parameterization	Dynamic at run-time	all basic types, all user-defined types, **address** type, **template.**
function (note 1)	Value, template, port and timer parameterization, type parameterization, signature parameterization	Dynamic at run-time	all basic types, all user-defined types, **address** type, **component** type, **port** type, **default**, **template** and **timer**.
altstep (note 1)	Value, template, port and timer parameterization, type parameterization, signature parameterization	Dynamic at run-time	all basic types, all user-defined types, **address** type, **component** type, **port** type, **default**, **template** and **timer**.
testcase (note 1)	Value, template, port and timer parameterization, type parameterization, signature parameterization	Dynamic at run-time	all basic types and of all user-defined types, **address** type, **template**.
signature	Value and template parameterization, type parameterization	Dynamic at run-time	all basic types, all user-defined types and **address** type, **component** type.
NOTE 1: Type and signature parameterization are always static at compile-time. NOTE 2: Only port and component types are parameterizable with signature formal parameters.			

Clause 5.4.1 Formal parameters

All types in TTCN-3 may be parameterized.

Clause 5.4.1.1 Formal parameters of kind value

In addition to the existing rules, TTCN-3 supports value parameterizations as follows:

- the value parameters of user-defined types shall be in parameters;
- the language element **signature** does not support *static* value parameterization.

Modify the text as follows:

Restriction a) is relaxed to:

a) Language elements which cannot be parameterized are: **const**, **var**, **timer**, **control**, ~~**record of**, **set of**, **enumerated**, **port**, **component** and sub-type definitions,~~ **group** and **import**.

Restriction e) is changed to:

e) The expression of formal parameter's default value has to be compatible with the type of the parameter. The expression may be any expression that is well-defined at the beginning of the scope of the parameterized entity, but shall not refer to other parameters of the same or any following parameter list.

Clause 5.4.1.2 Formal parameters of kind template

Restriction d) is changed to:

d) The default template instance has to be compatible with the type of the parameter. The template instance may be any template expression that is well-defined at the beginning of the scope of the parameterized entity, but shall not refer to other parameters in the same or any following parameter list.

Clause 5.4.1 Formal Parameters

Is extended by the following clause:

Clause 5.4.1.5 Formal parameters of kind type and signature

Type, template and behaviour definitions in TTCN-3 can have parameters of kind type.

Syntactical Structure

```
[ in ] [ TypeIdentifier | type | signature ] TypeParIdentifier [ ":=" ( Type | Signature ) ]
```

Semantic Description

Types and signatures passed into a parameterized object can be used inside the definition of that object. This includes the usage as type of value, template, and port parameters, as type of return values and within **`runs on`** and **`system`** clauses of behaviour definitions.

Any type and signature parameterization shall be resolved statically.

Type and signature parameters shall be written in a separate parameter list, enclosed in angle brackets.

Parameters of type kind may have a default type, which is given by a type assigned to the parameter. Similarly, parameters of signature kind may own a default signature, which is identified by assigning a signature to the parameter.

The actual parameters of a type parameter can be required to be subtype compatible with a specific type. This is indicated by referring to a specific type in the formal parameter list instead of using the keyword **`type`**.

Restrictions

a) Formal type and signature parameters shall be in parameters, which can optionally be indicated by the optional keyword **in**.

b) When a TypeIdentifier is used to specify the kind of the formal parameter, the default types shall be subtype compatible with the type of the parameter. For type compatibility see ETSI ES 201 873-1 [1] TTCN-3 Core Language, clause 6.3. The default type shall not refer to other type parameters in the same parameter list.

c) Void.

d) Void.

Examples

```
 // Definition of a list and a check function
type record of T MyList <in type T>;
function isElement <in type T>(in MyList<T> list, in T elem) return boolean { … }

// Definition of a protocol message
type record Data<in type PayloadType> {
  Header      hdr,
  PayloadType payload
}

// restricting the actual type parameters
// the function can create a component of a type that is an extension of CT.
type component CT { timer t_guard };
function MyFunction <in CT Comp> (in integer p) runs on CT {
  var Comp c := Comp.create;
  :
};

 function MyIntegerFunction<integer I := integer>(I p) return I {
    // actual parameter for I must be subtype compatible to integer
   :
 }
```

Clause 5.4.1.1 Formal Parameters of kind value

Formal parameters with default values are additionally restricted by:

Restrictions

Replace the text as follows:

e) ~~The expression of the default value has to be compatible with the type of the parameter. The expression shall not refer to elements of the component type of the optional **runs on** clause. The expression shall not refer to other parameters of the same parameter list. The expression shall not contain the invocation of functions with a **runs on** clause.~~

e) The type of a value parameter with a default value shall not be a type parameter.

Clause 5.4.1.2 Formal Parameters of kind template

Formal parameters with default templates are additionally restricted by:

Restrictions

a) ~~Only **function**, **testcase**, **altstep** and **template** definitions may have formal template parameters.~~

a) The type of a template parameter with a default template shall not be a type parameter.

Clause 5.4.2 Actual Parameters

Is extended by a new clause:

Clause 5.4.2.1 Actual type parameters

Types can be passed into parameterized TTCN-3 objects as actual type parameters. Signatures can be passed into parameterized TTCN-3 objects as actual signature parameters. Actual type and signature parameters can be provided both as a list in the same order as the formal parameters as well as in an assignment notation, explicitly using the associated formal parameter names. They can also be provided in mixed form, starting with actual parameters in list notation followed by additional ones in assignment notation.

Actual types can also be given as anonymous nested constructed types.

Syntactical Structure

```
[ TypeParIdentifier ":=" ] ( Type | Signature | NestedTypeDef )
```

Semantic Description

Actual type parameters that are passed to formal type parameters shall be types or formal type parameters or nested type definitions. Actual signature parameters that are passed to formal signature parameters shall be signatures or formal signature parameters. Any compatible type/signature can be passed as actual parameter, i.e. actual parameters are not limited to those types/signatures only known in the module containing the parameterized definition itself. Formal type and signature parameters passed as parameters are those from the enclosing scope unit.

An empty type/signature parameter list can be omitted in both the declaration and usage of that object.

Restrictions

a) When using list notation, the order of elements in the actual parameter list shall be the same as their order in the corresponding formal parameter list. For each formal parameter without a default type/signature there shall be an actual parameter. If a formal parameter with a default is followed by a formal parameter without a default, the actual parameter can be skipped by using dash "-" as actual parameter. If a formal parameter with a default is not followed by a parameter without a default, then the actual parameter can simply be omitted.

b) When using assignment notation, each formal parameter shall be assigned an actual parameter at most once. For each formal parameter without default-type or signature, there shall be an actual parameter. To use the default-type or signature of a formal parameter, no assignment for this specific parameter shall be provided.

c) If a formal parameter was defined using a specific component type, then the actual parameter shall be compatible with the type of the formal parameter. For type compatibility of component types see ETSI ES 201 873-1 [1], clause 6.3.3.

d) Instantiating formal type or signature parameters with actual types/signatures shall result in valid TTCN-3.

EXAMPLE 1: Type parameterization.

```
function f <in type T> (in T a, in T b) return T {
  return a + b}
var integer c := f<integer>(1, 2); // correct call, result 3
var integer c := f<boolean>(true, false); // incorrect
```

EXAMPLE 2: Signature parameterization.

```
type port P <signature S> procedure { inout S }

function g <signature S> (integer a, P<S> p) return integer {
  var integer result;
  p.call(s:{a}, 10.0) {
    [] p.getreply(s:{-}) -> value result {}
    [] p.catch(s, exc) { testcase.stop }
    [] p.catch(timeout) { testcase.stop }
  }
  return result;
}

template integer exc := ?;

signature s1(integer a) return integer exception(integer);

signature s2(float a) return float exception(boolean);

type component C<signature S> {
  port P<S> p;
}

function h() runs on C<s1> {
  var integer r := g<s1>(1, p) // correct
}

function h2() runs on C<s2> {
  var float r := g<s2>(1, p)    // incorrect; g returns integer and the actual call within
                       // function g also returns integer.
}
```

EXAMPLE 3: Parameterization with nested types.

```
type record of union {
    @default T val,
    record { T val, Annotation ann } annotated
} AnnotatedList<T>;

// actual parameters for formal parameter T can be anonymous nested type constructs
type AnnotatedList<record { integer a,
                            AnnotatedList<T := enumerated { a, b }> b
                   }> AnnotatedPairs;
```

5.3 Extension to ETSI ES 201 873-1, clause 6 (Types and values)

Clause 6.1.2.1 Nested type definitions for field types

Nested type definitions shall not have formal value parameters and shall not have formal type or signature parameters.

Clause 6 Types and values

Is extended by the following clause:

Clause 6.2.9

In the Semantic Description part, add the following:

In the lists indicating the allowed collection of (message) types or procedure signatures, actual type parameters for types with formal type parameters can be given as the keyword **all** to indicate that all instances of the referenced parameterized type for any data type are allowed to be sent or received (dependent on the direction of the list) over that port.

In restriction e) the following sentence is added:

MessageType can also be a generic data type instance where the actual formal type parameters can be replaced with the keyword **all**.

Clause 6.5 Value parameterization of types

TTCN-3 allows the use of value parameters in type definitions. This is applicable to all user-defined types, including subtypes of basic types, and excluding behaviour types and signatures.

The formal parameters may be used inside the type definition. The actual parameters have to be either formal value parameters of an enclosing type or they have to satisfy the same restrictions as global constants, see ETSI ES 201 873-1 [1] TTCN-3 Core Language clause 10.

When referring to a parameterized type, actual parameters have to be provided for each of the formal parameters of the type.

EXAMPLE 1: Length restriction.

```
type record length ( 0 .. maxAmount ) of float MyFloats ( in integer maxAmount );
const MyFloats(3) myConst := { 1.1, 2.1, 3.1 };
```

EXAMPLE 2: Range subtyping.

```
type integer MyInt ( in integer maxInt ) ( 0 .. maxInt );
const MyInt(127) myByte := 125;
const MyInt(127) myWord := 65335; // incorrect
```

EXAMPLE 3: Passing parameter.

```
type record MySquareIndex ( in integer maxInt ) {
   MyInt(maxInt) x,
   MyInt(maxInt) y
};
```

EXAMPLE 4: Generic type wildcard in port definition.

```
type record R<T> { record of charstring headers, T payload }

type port P message { inout R<all> }  // allows any instance of R with any data
                                      // type
```

5.4 Extension to ETSI ES 201 873-1, clause 8 (Modules)

Clause 8.2.3.1 General form of import

Import of definitions is additionally restricted by:

Restrictions

i) When importing a parameterized type the parameters are not resolved.

5.5 Extension to ETSI ES 201 873-1, annex A (BNF and static semantics)

The BNF is extended with the following clause and productions:

Clause A.1.6.1.14 Type parameter definitions

```
784001. FormalTypeParList ::= "<" FormalTypePar { "," FormalTypePar } ">"
784002. FormalTypePar ::= [ InParKeyword ] [ Type | SignatureKeyword | TypeDefKeyword ]
                      TypeParIdentifier  [ ":=" Type | Signature ]
784003. TypeParIdentifier ::= Identifier
784004. TypeActualParIdentifier ::= Identifier
784005. TypeParAssignment ::= TypeActualParIdentifier ":=" TypeActualPar
784006. ActualTypeParList ::= ( "<" ActualTypePar { "," ActualTypePar } ">" ) |
        ("<" ActualTypeParAssignment { "," ActualTypeParAssignment } ">")
784007. ActualTypePar ::= Type | Signature | "-" | AllKeyword
/* STATIC SEMANTICS The use of AllKeyword is only allowed inside PortDefAttribs */
784008. ActualTypeParAssignment ::= TypeActualParIdentifier ":=" ActualTypePar
784009. StructDefFormalParList ::= "(" StructDefFormalPar {"," StructDefFormalPar} ")"
784010. StructDefFormalPar ::=  FormalValuePar
784011. TypeActualParList ::= ( "(" TypeActualPar {"," TypeActualPar} ")" ) |
            ( "(" TypeParAssignment { "," TypeParAssignment } ")" )
784012. TypeActualPar ::= ConstantExpression | TypeActualParIdentifier
```

The following productions from ETSI ES 201 873-1 [1] TTCN-3 Core Language clause A.1.6 are modified as follows:

```
15. StructDefBody ::= (Identifier | AddressKeyword)
                      [FormalTypeParList] [StructDefFormalParList]
                      "{" [StructFieldDef {"," StructFieldDef}] "}"
27. UnionDefBody ::= (Identifier | AddressKeyword)
                     [FormalTypeParList] [StructDefFormalParList]
                     "{" UnionFieldDef {"," UnionFieldDef} "}"
33. StructOfDefBody ::= (Type | NestedTypeDef) (Identifier | AddressKeyword)
                        [FormalTypeParList] [StructDefFormalParList] [SubTypeSpec]
35. EnumDef ::= EnumKeyword (Identifier | AddressKeyword) [StructDefFormalParList]
                "{" EnumerationList "}"
41. SubTypeDef ::= Type (Identifier | AddressKeyword)
                   [FormalTypeParList] [StructDefFormalParList] [ArrayDef] [SubTypeSpec]
49. PortDefBody ::= Identifier [FormalTypeParList] [StructDefFormalParList] PortDefAttribs
73. ComponentDef ::= ComponentKeyword Identifier
                     [FormalTypeParList] [StructDefFormalParList]
                     [ExtendsKeyword ComponentType {"," ComponentType}]
                     "{" [ComponentDefList] "}"
76. ComponentType ::= ExtendedIdentifier
                      [ActualTypeParList] [ TypeActualParList ]
79. PortInstance ::= PortKeyword ExtendedIdentifier [ActualTypeParList]
                   [ TypeActualParList ] PortElement {"," PortElement}
86. BaseTemplate ::= (Type | Signature) Identifier
                     [FormalTypeParList] ["("TemplateOrValueFormalParList")"]
100. StructFieldRef ::= Identifier | PredefinedType | ReferencedType
/* STATIC SEMANTICS - PredefinedType and ReferencedType shall be used for anytype value notation
only. PredefinedType shall not be AnyTypeKeyword.*/
158. FunctionDef ::= FunctionKeyword Identifier [FormalTypeParList]
                     "("[FunctionFormalParList] ")" [RunsOnSpec] [ReturnType] StatementBlock
175. FunctionInstance ::= FunctionRef [ActualTypeParList] "(" [ActualParList] ")"
178. SignatureDef ::= SignatureKeyword Identifier [FormalTypeParList]
                      "("[SignatureFormalParList] ")" [ReturnType | NoBlockKeyword]
                      [ExceptionSpec]
183. Signature ::= ExtendedIdentifier [ActualTypeParList]
185. TestcaseDef ::= TestcaseKeyword Identifier [FormalTypeParList]
                     "("[TemplateOrValueFormalParList] ")" ConfigSpec
                     StatementBlock
190. TestcaseInstance ::= ExecuteKeyword "(" ExtendedIdentifier [ActualTypeParList]
                          "(" [ActualParList] ")"
                          ["("(Expression | Minus) ["," SingleExpression]] ")"
192. AltstepDef ::= AltstepKeyword Identifier [FormalTypeParList]
                    "("[FunctionFormalParList] ")" [RunsOnSpec]
                    "{" AltstepLocalDefList AltGuardList "}"
196. AltstepInstance ::= ExtendedIdentifier [ActualTypeParList] "(" [ActualParList] ")"
408. ReferencedType ::= [GlobalModuleId Dot] TypeReference [TypeActualParList]
                        [ExtendedFieldReference]

541. ExtendedFieldReference ::= {((Dot (Identifier [ActualTypeParList][TypeActualParList]|
                                        PredefinedType ))
                                   | ArrayOrBitRef_| ("["Minus "]") ) }+
```

```
/* 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.*/
```

5.6 Extension to ETSI ES 203 790, clause 5.1 (Classes and Objects)

Clause 5.1.1.0 General

Change the following paragraph:

Syntactical Structure

Change to the following:

```
[public | private]
type [external] class [@final |@abstract]
Identifier [FormalTypeParList] [extends Type]
[runsOnSpec] [systemSpec] [mtcSpec]
"{" {ClassMember} "}"
[finally StatementBlock]
```

Semantic Description

Add the following:

Class definitions may be parameterized with formal type parameters.

Classes defined with formal type parameters can be referenced only by providing an actual type parameter list compatible with the formal type parameter list of the class.

Examples

```
type external class @abstract Iterator<T> {
  function hasNext() return boolean;
  function next() return T;
}

type external class @abstract Container<T> {
  function iterator() return Iterator<T>;
  function size() return integer;
  function isEmpty() return boolean;
  function add(T p_element);
  function contains(T p_element) return boolean;
}

type external class Set<T> extends Container<T> {
}

type external class List<T> extends Container<T> {
  function getElement(integer p_pos) return T;
}

type external class MapEntr<Key, Value> {
  function getKey() return Key;
  function getValue() return Value;
}

type external class Map<Key,Value> {
  function get(Key p_key) return Value;
  function put(Key p_key, Value val);
  function keySet() return Set<Key>;
  function values() return List<Value>;
  function entrySet() return Set<MapEntry<Key, Value>>;
  function containsKey(Key p_key) return boolean;
  function contains(Value p_val) return boolean;
}

var Map<charstring, integer> v_map := Map<charstring, integer>.create();
var Container<charstring> v_container := …
```

```
select class (v_container) {
  case (List<charstring>) { … }
  case (Set<charstring>) { … }
}
```

5.7 Extension to ETSI ES 203 790, clause A.3 (Additional TTCN-3 syntax BNF productions)

The following BNF rules are changed:

```
033001. ClassDef ::= [ ExtKeyword ] ClassKeyword [ FinalModifier | AbstractModifier ]
                     Identifier [ FormalTypeParList ] [ ExtendsKeyword Type ]
                     [ RunsOnSpec ] [ MtcSpec ] [ SystemSpec ]
                     "{" ClassMemberList "}"
                     [ FinallyKeyword BasicStatementBlock ]
```

6 Package semantics

6.0 General

The semantics of a declaration with type parameters is defined only for the instantiations of the declaration, i.e. only the instances of the declaration with actual types provided are considered meaningful.

6.1 Extension to ETSI ES 201 873-4, clause 6 (Restrictions)

Value parameterization of types and general type parameterization are static aspects. They are not relevant for the operational semantics.

The operational semantics therefore does not provide:

a) A semantics for type parameterization. The instantiation of type parameters has to be done in the definitions part of a TTCN-3 module. The operational semantics is defined for the instantiated definitions only.

b) A semantics for value parameterization of types. The instantiation of value parameters of types has to be done in the definitions part of a TTCN-3 module. The operational semantics is defined for the instantiated type definitions only.

7 TRI and Extended TRI extensions for the package

7.1 Extension to ETSI ES 201 873-5

Advanced parameterization has no effects on the TRI.

7.2 Extension to ETSI ES 202 789, clause 7 (TRI extensions for the package)

Clause 5.6.3.1 xtriExternalFunction

The description of the operation is modified as follows:

Signature	`TriStatusType xtriExternalFunction(`
	`in TriFunctionIdType functionId,`
	`inout TciParameterListType parameterList,`
	`out Value returnValue)`

In Parameters	`functionId` identifier of the external function
Out Parameters	`returnValue` (optional) encoded return value
InOutParameters	`parameterList` a list of ~~encoded~~ parameters for the indicated function. The parameters in `parameterList` are ordered as they appear in the TTCN-3 function declaration.
Return Value	The return status of the `triExternalFunction` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.
Constraints	This operation is called by the TE when it executes a function which is defined to be TTCN-3 external (i.e. all non-external functions are implemented within the TE). No error shall be indicated by the PA in case the value of any *out* parameter is non-null.
Effect	For each external function specified in the TTCN-3 ATS the PA shall implement the behaviour. On invocation of this operation the PA shall invoke the function indicated by the identifier `functionId`. It shall access the specified *in* and *inout* function parameters in `parameterList`, evaluate the external function using the values of these parameters, and compute values for *inout* and *out* parameters in `parameterList`. The operation shall then return values for all *inout* and *out* function parameters and the return value of the external function. If no return type has been defined for this external function in the TTCN-3 ATS, the distinct value `null` shall be used for the latter. The `triExternalFunction` operation returns ***TRI_OK*** if the PA completes the evaluation of the external function successfully, ***TRI_Error*** otherwise. Note that whereas all other TRI operations are considered to be non-blocking, the `triExternalFunction` operation is considered to be *blocking.* That means that the operation shall not return before the indicated external function has been fully evaluated. External functions have to be implemented carefully as they could cause deadlock of test component execution or even the entire test system implementation. In case that the external function is defined with formal type parameters, the first items in the parameterList will be representations of the actual type parameters given to the invocation of the external function, using TypeValue instances that contain the actual type values. Thus, the parameterList is a concatenation of the actual type parameter list and the actual value parameter list of the invocation of the external function in the order of the specification of the formal parameter lists of the declaration of the external function.

8 TCI extensions for the package

8.1 Extension to ETSI ES 201 873-6, clause 7 (TTCN 3 control interface and operations)

Clause 7.2.1.2 Communication types

The `TciBehaviourIdType` is extended to cover also the actual type parameters.

`TciBehaviourIdType` A value of type `TciBehaviourIdType` identifies a TTCN-3 behaviour functions, including actual type parameters.

Clause 7.2.2.1 Abstract TTCN-3 data types

The following additional operations are defined for the abstract data type `Type`:

`TciParameterListType getValueParameters()`	Returns a list of the actual value parameters of this type. The list can be empty.
`TciParameterTypeListType getTypeParameters()`	Returns a list of the actual type parameters of this type. The list can be empty.

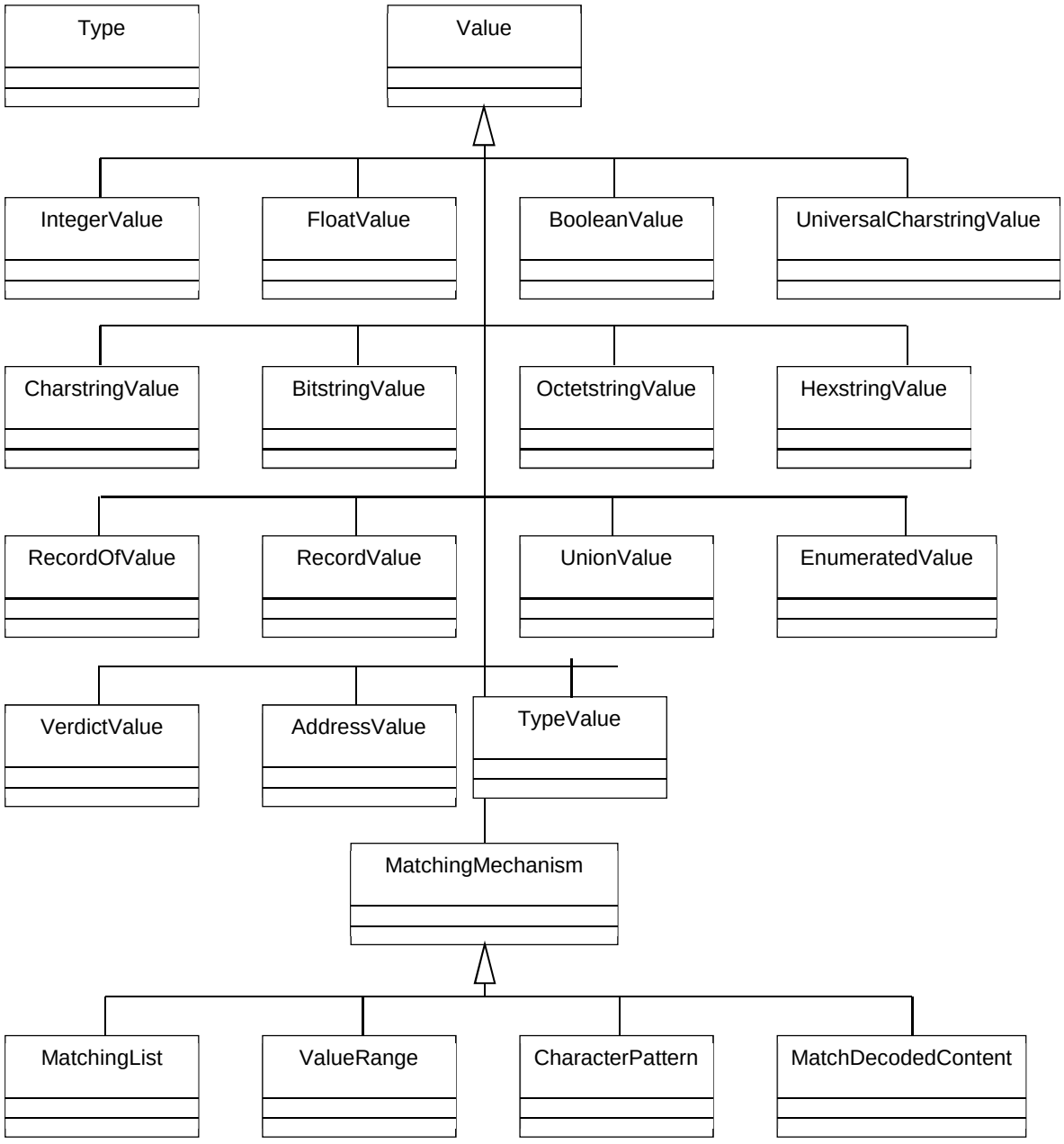
The list of operations of the class Type in **figure 4 Hierarchy of abstract values** is extended by:

getValueParameters():TciTypes::TciParameterListType

getTypeParameters():TciTypes::TciParameterTypeListType

Clause 7.2.2.2.0 Basic rules

A new `TypeValue` subclass of the class `Value` is added to the diagram.



The following TypeValue abstract data type is added:

Clause 7.2.2.2.16 The abstract data type TypeValue

The abstract data type TypeValue is based on the abstract data type Value. Is is used to represent an actual type parameter. It specifies how to get and set the actual type of the parameter.

The following operations are defined on the abstract data type TypeValue:

`Type getActualType()`

Returns the actual type of the formal type parameter.

`void setActualType(in Type typeInst)`

Sets the actual type of the formal type parameter.

Clause 7.3.1.1.7 tciStartTestCase

The usage of tciStartTestCase is additionally constrained by:

tciStartTestCase shall not be used for testcases with type parameters.

8.2 Extension to ETSI ES 201 873-6, clause 8 (Java™ language mapping)

Clause 8.2.2.6 TciBehaviourIdType

Text is modified as follows:

TciBehaviourIdType is mapped to the following interface, providing access to the name as well as to the type parameters of the behaviour.

```
// TCI IDL TciBehaviourIdType
package org.etsi.ttcn.tci;
public interface TciBehaviourId extends QualifiedName {
public TciParameterTypeList         getTypeParameters();
}
```

Methods:

- getTypeParameters() Returns the list of the actual type parameters of this behaviour.

Clause 8.2.3.1 Type

The interface Type is extended by the two new operations:

```
// TCI IDL Type
package org.etsi.ttcn.tci;
public interface Type {
    public TciModuleId  getDefiningModule ();
    public String       getName ();
    public int          getTypeClass ();
    public Value        newInstance ();
    public TciParameterList     getValueParameters();
    public TciParameterTypeList getTypeParameters();
    public String       getTypeEncoding ();
    public String       getTypeEncodingVariant();
    public String[]     getTypeExtension();
}
```

Methods:

`getValueParameters()` Returns the list of the actual value parameters of this type.

`getTypeParameters()` Returns the list of the actual type parameters of this type.

Clause 8.3.2.4 TciTypeClassType

The clause is to be modified:

`TciTypeClassType` is mapped to the following interface:

```
// TCI IDL TciTypeClassType
package org.etsi.ttcn.tci;
public interface TciTypeClass {
    public final static int ADDRESS             = 0 ;
    public final static int ANYTYPE             = 1 ;
    public final static int BITSTRING               = 2 ;
    public final static int BOOLEAN             = 3 ;
    public final static int CHARSTRING              = 5 ;
    public final static int COMPONENT               = 6 ;
    public final static int ENUMERATED              = 7 ;
    public final static int FLOAT                   = 8 ;
    public final static int HEXSTRING               = 9 ;
    public final static int INTEGER             = 10 ;
    public final static int OCTETSTRING         = 12 ;
    public final static int RECORD                  = 13 ;
    public final static int RECORD_OF               = 14 ;
    public final static int ARRAY                   = 15 ;
    public final static int SET                 = 16 ;
    public final static int SET_OF                  = 17 ;
    public final static int UNION                   = 18 ;
    public final static int UNIVERSAL_CHARSTRING    = 20 ;
    public final static int VERDICT             = 21 ;
    public final static int DEFAULT             = 22 ;
    public final static int PORT                    = 23 ;
    public final static int TIMER                   = 24 ;
    public final static int TYPE                    = 30 ;
```

Clause 8.3.4.16 TypeValue

The clause is to be added.

`TypeValue` is mapped to the following interface:

```
// TCI IDL RecordValue
package org.etsi.ttcn.tci;
public interface TypeValue {
    public Type     getActualType() ;
    public void     setActualType(Type type) ;
}
```

Methods:

- `getActualType` Returns the actual type assigned to this formal type parameter.
- `setActualType` Set the actual type of this formal type parameter.

8.3 Extension to ETSI ES 201 873-6, clause 9 (ANSI C language mapping)

Clause 9.2 Value interfaces

The interface of Type is extended by two additional operations. In addition to that, two functions are added for the new `TypeValue` abstract data type.

TCI IDL Interface	ANSI C representation	Notes and comments
Type		
`TciParameterListType getValueParameters()`	`TciParameterListType tciGetValueParameters(Type inst)`	
`TciParameterTypeListType getTypeParameters()`	`TciParameterTypeListType tciGetTypeParameters(Type inst)`	
TypeValue		
Type getActualType()	`Type tciGetTypeValueType (TypeValue inst)`	
void setActualType (in Type value)	`void tciSetTypeValueType (TypeValue inst, Type value)`	

Clause 9.5 Value interfaces

`TciBehaviourId` is extended to hold the actual type parameters.

TCI IDL ADT	ANSI C representation (Type definition)	Notes and comments
`TciBehaviourIdType`	`QualifiedName`	The field `aux` holds a reference to an element of type `TciParameterTypeListType`
`TciTypeClassType`	`typedef enum { TCI_ADDRESS_TYPE = 0, TCI_ANYTYPE_TYPE = 1, TCI_BITSTRING_TYPE = 2, TCI_BOOLEAN_TYPE = 3, TCI_CHARSTRING_TYPE = 5, TCI_COMPONENT_TYPE = 6, TCI_ENUMERATED_TYPE = 7, TCI_FLOAT_TYPE = 8, TCI_HEXSTRING_TYPE = 9, TCI_INTEGER_TYPE = 10, TCI_OCTETSTRING_TYPE = 12, TCI_RECORD_TYPE = 13, TCI_RECORD_OF_TYPE = 14, TCI_ARRAY_TYPE = 15, TCI_SET_TYPE = 16, TCI_SET_OF_TYPE = 17, TCI_UNION_TYPE = 18, TCI_UNIVERSAL_CHARSTRING_TYPE = 20, TCI_VERDICT_TYPE = 21 TCI_DEFAULT_TYPE = 22, TCI_PORT_TYPE = 23, TCI_TIMER_TYPE = 24, TCI_TYPE_TYPE = 30 } TciTypeClassType;`	

8.4 Extension to ETSI ES 201 873-6, clause 10 (C++ language mapping)

Clause 10.5.2.1 TciBehaviourId

Identifies a TTCN-3 behaviour functions. It is mapped to the following pure virtual class:

```
class TciBehaviourId: ORG_ETSI_TTCN3_TRI::QualifiedName {
public:
    virtual ~TciBehaviourId ();
    virtual TciParameterTypeList getTypeParameters() const =0;
    virtual Tboolean equals (const TciBehaviourId &bid) const =0;
    virtual TciBehaviourId * cloneBehaviourId () const =0;
    virtual Tboolean operator< (const TciBehaviourId &bid) const =0;
}
```

Clause 10.5.2.1.1 Methods

To be extended by the following text:

`getTypeParameters()` Returns the list of the actual type parameters of this behaviour.

Clause 10.5.3.1 TciType

The interface Type is extended by the two new operations `getValueParameters` and `getTypeParameters`:

```
class TciType {
public:
    virtual ~TciType ();
    virtual const TciModuleId & getDefiningModule () const =0;
    virtual const Tstring & getName () const =0;
    virtual const TciTypeClass & getTypeClass () const =0;
    virtual const Tstring & getTypeEncoding () const =0;
    virtual const Tstring & getTypeEncodingVariant () const =0;
    virtual TciValue * newInstance ()=0;
    virtual const TciParameterList & getValueParameters()const =0;
    virtual const TciParameterTypeList & getTypeParameters()const =0;
    virtual Tboolean equals (const TciType &typ) const =0;
    virtual TciType * cloneType () const =0;
    virtual Tboolean operator< (const TciType &typ) const =0;
}
```

Clause 10.5.3.1.1 Methods

To be extended by the following text:

`getValueParameters()` Returns the list of the actual value parameters of this type.

`getTypeParameters()` Returns the list of the actual type parameters of this type.

Clause 10.5.3.23 TypeValue

TTCN-3 actual type support. It is mapped to the following pure virtual class:

```
class TypeValue : public virtual TciValue {
public:
    virtual ~TypeValue ();
    virtual TciType &getActualType () =0;
    virtual void setActualType (const TciType &p_new_type)=0;
}
```

Methods:

`~TypeValue`
: Destructor

`getActualType`
: Return the actual type of the formal type parameter.

`setActualType`

Set the actual type of the formal type parameter.

8.5 Extension to ETSI ES 201 873-6, annex A (IDL Specification of TCI)

The definitions of `TciBehaviourIdType` and `Type` are extended to provide also actual type parameters:

```
  struct TciBehaviourIdType {
    TString moduleName;
    TString baseName;
    TciParameterTypeListType typeParameters;
  };

// Abstract data type "Type"
  interface Type {
    TciModuleIdType  getDefiningModule ();
    TString          getName ();
    TciTypeClassType getTypeClass ();
    Value            newInstance ();
    TciParameterListType getValueParameters();
    TciParameterTypeListType getTypeParameters()
    TString          getTypeEncoding ();
    TString          getTypeEncodingVariant ();
    TStringSeq       getTypeExtension ();
  };
```

8.6 Extension to ETSI ES 201 873-6, clause 12 (C# Mapping Specification of TCI)

New `TypeValue` abstract data type mapping is added:

Clause 12.4.4.10 TypeValue

`TypeValue` is mapped to the following interface:

```
public interface ITciTypeValue : ITciValue {
    ITciType ActualType { get; set; }
}
```

Members:

- `ActualType`
  Gets or sets the actual type of the formal type parameter.

9 Package Extensions for the use of ASN.1 with TTCN-3

9.1 Extension to ETSI ES 201 873-7, clause 10 (Parameterization in ASN.1)

Clause 10 Parameterization in ASN.1

Is replaced by:

Parameterization of ASN.1 definitions is specified by Recommendation ITU-T X.683 [10].

ASN.1 **type**,value and valueset definitions, parameterized merely with ASN.1 type, value and valueset formal parameters (i.e. when all formal parameter(s) have the form "*DummyType*", "*DummyType: dummyvalue*", *"Type: dummyvalue"*, "*DummyType:Dummyvalueset*" or "*Type:Dummyvalueset*"), can be imported to TTCN-3 and used to define TTCN-3 definitions. When importing all definitions of an ASN.1 module, such parameterized definitions shall also be imported.

When translating parameterized ASN.1 definitions, the following rules apply, in addition to those given in clause 9 of ETSI ES 201 873-7 [5]:

a) In this clause the following editorial conventions are used:

- visible defined ASN.1 types used as governors in formal parameters are referred to as *Type;*
- ASN.1 names which are not identifiers of visible defined types but used as governors in formal parameters in the role of types, are referred to as *DummyType*;
- ASN.1 names which are used as formal parameters identifiers in the role of values, are referred to as *dummyvalue*;
- ASN.1 names which are used as formal parameters identifiers in the role of valuesets, are referred to as *Dummyvalueset*.

NOTE 1: Please note that according to clause 8.4 of Recommendation ITU-T X.683 [10] "*dummyvalue*" and "*Dummyvalueset*" hides any other ASN.1 definition with the same name within parameterized definitions (including the parameter list itself) in any given instantiation.

b) For the ASN.1 formal parameters in the list using the form of "*DummyType*" and "*DummyType:dummyvalue*", for each *DummyType* with distinct names one TTCN-3 type formal parameter shall be created. The names of the formal parameters shall be the ASN.1 name of the corresponding *DummyType*, converted according to clause 8.2 of ETSI ES 201 873-7 [5] and the type of each parameter shall be **`type`**.

c) For each value formal parameter in the form of "*DummyType:dummyvalue*" and "*Type:dummyvalue*", one TTCN-3 value formal parameter shall be created. The name of the value parameter shall be the ASN.1 name of the *dummyvalue*, converted according to clause 8.2 of ETSI ES 201 873-7 [5]. The type of the TTCN-3 value formal parameter shall be:

- the TTCN-3 formal type parameter created for the given *DummyType* in the case of "*DummyType:dummyvalue*" ASN.1 parameters; and
- the TTCN-3 type, associated with the given ASN.1 type, in case of "*Type:dummyvalue*" ASN.1 parameters.

d) For each value set formal parameter in the form of "*DummyType:Dummyvalueset*" and "*Type:Dummyvalueset*", one TTCN-3 type formal parameter shall be created. The name of the type parameter shall be the ASN.1 name of the *Dummyvalueset*, converted according to clause 8.2 of ETSI ES 201 873-7 [5]. The type of the TTCN-3 type formal parameter shall be:

- **`type`** in the case of "*DummyType:Dummyvalueset* " ASN.1 parameters and the TTCN-3 type, associated with the given ASN.1 type, in case of "*Type:Dummyvalueset*" ASN.1 parameters.

All references to parameterized ASN.1 definitions shall be resolvable at compile-time. The type of the actual parameters passed to type parameters and type-parameterized value parameters shall be resolvable at compile-time.

When the translated parameterized definitions are referred to in TTCN-3, the consistency of the definition with clause 6 of ETSI ES 201 873-1 [1] shall be checked after resolving the actual parameters. If a type or value parameter is used to constrain a type parameter, and the actual type parameter to be constrained is a structured type, this type constraint shall be considered as a newly constrained subtype, i.e. it shall be allowed (see note to table 3 of ETSI ES 201 873-1 [1] and the examples below).

NOTE 2: As not all ASN.1 type constrains can be translated to TTCN-3 (see clause 9.1 of ETSI ES 201 873-7 [5]), users should be cautious to preserve the original intention of the ASN.1 specification, when using parameterized ASN.1 definitions from TTCN-3.

EXAMPLES:

-- the ASN.1 module

```
    MyASN1-ParameterizedStuff DEFINITIONS ::=
    BEGIN
        Type-Parameterized-ASN1-type { MyParType } ::= SEQUENCE {
            id      INTEGER,
            message     MyParType
        }

        Value-Parameterized-ASN1-type {
            INTEGER:myIntParam, MyParType, MyParType:firstParamValue, MyParType:secondParamValue
        } ::= SEQUENCE
        {
            id      INTEGER (0.. myIntParam),
            message     MyParType ( firstParamValue | secondParamValue )
        }

        ValueSet-Parameterized-ASN1-type {
            INTEGER:MyIntListParam, MyParType, MyParType:MyValueListParam
        } ::= SEQUENCE
        {
            id      INTEGER ( 0 | MyIntListParam ),
            message     MyParType ( MyValueListParam )
        }
    END

// Will be translated to the following associated TTCN-3 module and declarations:
    module MyASN1_ParameterizedStuff
    {
        type_record Type_Parameterized_ASN1_type
          < type MyParType >
        {
            integer     value,
            MyParType   message
        }

        type record Value_Parameterized_ASN1_type
          < type MyParType >
          (integer myIntParam, MyParType firstParamValue, MyParType secondParamValue)
        {
            integer     id (0.. myIntParam),
            MyParType   message     (firstParamValue, secondParamValue)
        }

        type record ValueSet_Parameterized_ASN1_type
          < integer MyIntListParam, type MyParType, type MyValueListParam >{
        {
            integer     id      ( 0 , MyIntListParam ),
            MyParType   message     ( MyValueListParam )
        }
    }

// And the above definitions can be used like below
    module Using_MyASN1_ParameterizedStuff {
        import from MyASN1_ParameterizedStuff language "ASN.1:2008" all;

        //auxiliary type definitions
        type integer Integer;

        type integer OneorTwo ( 1, 2 );

        type record MyRecord { integer field1, integer field2 optional }

        type MyRecord MyBroaderRestrictedRecord ( {0,omit}, {1,omit}, {2,omit}, {1,2} )

        type MyBroaderRestrictedRecord MyRestrictedRecord ( {0,omit},{1,2} )

        //uses of the imported ASN.1 definitions
        const ValueSet_Parameterized_ASN1_type
          < OneorTwo, MyRecord, MyRestrictedRecord > MyConst1 := {
            id := 0,
            // any other value than 0, 1 or 2 should cause an error
            message := { field1 := 0, field2 := omit }
            // any other value than {0,omit} or {1,2} should cause an error
        }
```

```
    const ValueSet_Parameterized_ASN1_type
      < OneorTwo, integer, OneorTwo > MyConst2 := {
        id := 0,
        // any other value than 0, 1 or 2 should cause an error
        message := 2
        // any other value than 0, 1 or 2 should cause an error
    }

    const ValueSet_Parameterized_ASN1_type
      < OneorTwo, OneorTwo, OneorTwo > MyConst3 := { 0, 2 }
    // MyConst3 has the same value as MyConst2 (their type definitions are practically equal)

    const ValueSet_Parameterized_ASN1_type
      < OneorTwo, MyBroaderRestrictedRecord, MyRestrictedRecord > MyConst4 := {
        id := 0,
        message := {1,2}
    }
    // the type of MyConst4 practically denotes the same value set as the type of MyConst1

    const ValueSet_Parameterized_ASN1_type < integer , ... > MyConst5 := ...
    // This is OK as the built-in type integer is a subtype of itself
    // (the first type parameter is used to constrain the field "id" within the type definition)
    // the here the id field would not be constrained by the parameter.

    const ValueSet_Parameterized_ASN1_type < Integer , ... > MyConst6 := ...
    // This is OK as Integer is defined as subtype of integer,
    // even if they denote the same value set.

    const ValueSet_Parameterized_ASN1_type
      < OneorTwo, MyRestrictedRecord, MyBroaderRestrictedRecord > MyConst7 := ...
    // causes an error as MyBroaderRestrictedRecord is not a subtype of MyRestrictedRecord
    // (the third type parameter is used to constrain the second type parameter in the field
    // "message")

}
```

All other parameterized ASN.1 definitions, not specified in the previous paragraphs of this clause, shall not be imported to and referenced in TTCN-3 modules.

For example the following is not legal because it would associate a TTCN-3 type which should take an ASN.1 object set as an actual parameter.

```
    MyASN1module DEFINITIONS ::=
    BEGIN
        -- ASN.1 Module definition

    -- Information object class definition
    MESSAGE ::= CLASS { &msgTypeValue   INTEGER UNIQUE,
                                        &MsgFields}

    -- Information object definition
    setupMessage MESSAGE ::= {  &msgTypeValue       1,
                                &MsgFields          OCTET STRING}

    setupAckMessage MESSAGE ::= {   &msgTypeValue   2,
                                    &MsgFields      BOOLEAN}

    -- Information object set definition
    MyProtocol MESSAGE ::= {setupMessage | setupAckMessage}

    -- ASN.1 type constrained by object set
    MyMessage{ MESSAGE : MsgSet} ::= SEQUENCE
    {
        code    MESSAGE.&msgTypeValue({MsgSet}),
        Type MESSAGE.&MsgFields({MsgSet})
    }
END

module MyTTCNModule
{
    // TTCN-3 module definition
    import from MyASN1module language "ASN.1:2002" all;

    // Illegal TTCN-3 type with object set as parameter
    type record Q(MESSAGE MyMsgSet) ::= {   Z                   field1,
                                            MyMessage(MyMsgSet) field2}
}
```

To make this a legal definition the extra ASN.1 type My Message1 has to be defined as shown below. This resolves the information object set parameterization and can therefore be directly used in the TTCN-3 module.

```
MyASN1module DEFINITIONS ::=
BEGIN
    -- ASN.1 Module definition

    …

    MyProtocol MESSAGE ::= {setupMessage | setupAckMessage}

    -- Extra ASN.1 type to remove object set parameterization
    MyMessage1 ::= MyMessage{MyProtocol}
END

module MyTTCNModule
{
    // TTCN-3 module definition
    import from MyASN1module language "ASN.1:2002" all;

    // Legal TTCN-3 type with no object set as parameter
    type record Q := {  Z              field1,
                        MyMessage1     field2}
}
```

10 Documentation extensions for the package

10.1 Extension to ETSI ES 201 873-10, clause 6 (Tagged paragraphs)

Clause 6.6 The @param tag

The @param tag is applicable to all parameterized TTCN-3 definitions.

10.2 Extension to ETSI ES 201 873-10, annex A (where Tags can be used)

Table A.1: "Relation of documentation tags and TTCN-3" is modified as follows (only the `@param` row is changed)

	Simple Data Types	Structured Data Types	Component Types	Port Types	Modulepars	Constants	Templates	Signatures	Functions (TTCN-3 and external)	Altsteps	Test Cases	Modules	Groups	Control Parts	Component local definitions	Used in implicit form (see clause 7)	Embedded in other tags
`@author`	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
`@config`											X						
`@desc`	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
`@exception`								X								X	
`@member`		X[1]	X	X	X[1]	X[1]	X[1]									X	
`@param`	X	X	X	X	X	X	X	X	X	X	X					X	
`@priority`											X						
`@purpose`											X	X					
`@remark`	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
`@reference`	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
`@requirement`									X	X	X	X					
`@return`								X	X							X	
`@see`	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X
`@since`	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
`@status`	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
`@url`	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X
`@verdict`									X	X	X	X					
`@version`	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

NOTE: [1] Preceding language elements of record, set, union or enumerated types only.

History

Document history		
V1.1.1	July 2009	Publication
V1.2.1	May 2011	Publication
V1.3.1	April 2013	Publication
V1.4.1	June 2014	Publication
V1.5.1	June 2015	Publication
V1.6.1	April 2017	Publication
V1.7.1	February 2020	Membership Approval Procedure MV 20200428: 2020-02-28 to 2020-04-28
V1.7.1	May 2020	Publication