

# ETSI ES 202 784 V1.1.1 (2009-07)

---

*ETSI Standard*

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

---



---

**Reference**DES/MTS-00123 T3Ext\_AdvParam

---

---

**Keywords**conformance, testing, TTCN

---

**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**

---

Individual copies of the present document can be downloaded from:

<http://www.etsi.org>

The present document may be made available in more than one electronic version or in print. In any case of existing or perceived difference in contents between such versions, the reference version is the Portable Document Format (PDF). In case of dispute, the reference shall be the printing on ETSI printers of the 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:

[http://portal.etsi.org/chaicor/ETSI\\_support.asp](http://portal.etsi.org/chaicor/ETSI_support.asp)

---

**Copyright Notification**

---

No part may be reproduced except as authorized by written permission.  
The copyright and the foregoing restriction extend to reproduction in all media.

© European Telecommunications Standards Institute 2009.  
All rights reserved.

**DECT™**, **PLUGTESTS™**, **UMTS™**, **TIPHON™**, the TIPHON logo and the ETSI logo are Trade Marks of ETSI registered for the benefit of its Members.

**3GPP™** is a Trade Mark of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners.

**LTE™** is a Trade Mark of ETSI currently being 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.

---

# Contents

Intellectual Property Rights .....	4
Foreword.....	4
1 Scope .....	5
2 References .....	5
2.1 Normative references .....	5
2.2 Informative references.....	6
3 Definitions and abbreviations.....	6
3.1 Definitions.....	6
3.2 Abbreviations .....	6
4 Package conformance and compatibility.....	6
5 Package concepts for the core language.....	7
5.1 Extension to ES 201 873-1, clause 4 (Introduction).....	7
5.2 Extension to ES 201 873-1, clause 5 (Basic language elements) .....	7
5.3 Extension to ES 201 873-1, clause 6 (Types and values).....	11
5.4 Extension to ES 201 873-1, clause 8 (Modules).....	11
5.5 Extension to ES 201 873-1, annex A (BNF and static semantics).....	12
6 Package semantics.....	13
6.1 Extension to ES 201 873-4, clause 6 (Restrictions) .....	13
7 TRI extensions for the package .....	13
8 TCI extensions for the package .....	13
8.1 Extension to ES 201 873-6, clause 7 (TTCN 3 control interface and operations).....	13
8.2 Extension to ES 201 873-6, clause 8 (Java language mapping) .....	14
8.3 Extension to ES 201 873-6, clause 9 (ANSI C language mapping) .....	15
8.4 Extension to ES 201 873-6, clause 10 (C++ language mapping) .....	15
8.5 Extension to ES 201 873-6, annex A (IDL Specification of TCI).....	16
9 Package Extensions for the use of ASN.1 with TTCN-3 .....	16
9.1 Extension to ES 201 873-7, clause 10 (Parameterization in ASN.1).....	16
10 Documentation extensions for the package.....	17
10.1 Extension to ES 201 873-10, clause 6 (Tagged paragraphs) .....	17
10.2 Extension to ES 201 873-10, annex A (where Tags can be used) .....	18
History .....	19

---

## Intellectual Property Rights

IPRs essential or potentially essential to the present document 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 (<http://webapp.etsi.org/IPR/home.asp>).

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.

---

## Foreword

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

The present document relates to the multi-part ES 201 873 standard covering the Testing and Test Control Notation version 3, as identified below:

- Part 1: "TTCN-3 Core Language";
- Part 2: "TTCN-3 Tabular presentation Format (TFT)";
- Part 3: "TTCN-3 Graphical presentation Format (GFT)";
- Part 4: "TTCN-3 Operational Semantics";
- Part 5: "TTCN-3 Runtime Interface (TRI)";
- Part 6: "TTCN-3 Control Interface (TCI)";
- Part 7: "Using ASN.1 with TTCN-3";
- Part 8: "The IDL to TTCN-3 Mapping";
- Part 9: "Using XML schema with TTCN-3";
- Part 10: "TTCN-3 Documentation Comment Specification".

---

# 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

References are either specific (identified by date of publication and/or edition number or version number) or non-specific.

- For a specific reference, subsequent revisions do not apply.
- Non-specific reference may be made only to a complete document or a part thereof and only in the following cases:
  - if it is accepted that it will be possible to use all future changes of the referenced document for the purposes of the referring document;
  - for informative references.

Referenced documents which are not found to be publicly available in the expected location might be found at <http://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.

## 2.1 Normative references

The following referenced documents are indispensable for the application of the present document. For dated references, only the edition cited applies. For non-specific references, the latest edition of the referenced document (including any amendments) applies.

- [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".

## 2.2 Informative references

The following referenced documents are not essential to the use of the present document but they assist the user with regard to a particular subject area. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Not applicable.

---

## 3 Definitions and abbreviations

### 3.1 Definitions

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

**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 Abbreviations

For the purposes of the present document, the abbreviations given in ES 201 873-1 [2], ES 201 873-4 [2], ES 201 873-5 [3], ES 201 873-6 [4], ES 201 873-7 [5], 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-3:2009 Advanced Parameterization" - to be used with modules complying with the present document.

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 and in ES 201 873-1 [1] and ES 201 873-4 [2].

The package presented in this document is compatible to:

- ES 201 873-1, version 4.1.1;
- ES 201 873-2, version 3.2.1;
- ES 201 873-3, version 3.2.1;
- ES 201 873-4, version 4.1.1;

ES 201 873-5, version 4.1.1;

ES 201 873-6, version 4.1.1;

ES 201 873-7, version 4.1.1;

ES 201 873-8, version 3.3.1;

ES 201 873-9, version 4.1.1;

ES 201 873-10, version 3.4.1.

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 ES 201 873-1, clause 4 (Introduction)

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

- type parameterization.

### 5.2 Extension to 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 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
<b>module</b>	Value parameterization	Static at start of run-time	all basic types, all user-defined types and <b>address</b> type.
<b>type</b>	Value parameterization, type parameterization	Static at compile-time	all basic types, all user-defined types and <b>address</b> type.
<b>template</b>	Value and template parameterization, type parameterization	Dynamic at run-time	all basic types, all user-defined types, <b>address</b> type, <b>template</b> .
<b>function</b>	Value, template, port and timer parameterization, type parameterization	Dynamic at run-time	all basic types, all user-defined types, <b>address</b> type, <b>component</b> type, <b>port</b> type, <b>default</b> , <b>template</b> and <b>timer</b> .
<b>altstep</b>	Value, template, port and timer parameterization, type parameterization	Dynamic at run-time	all basic types, all user-defined types, <b>address</b> type, <b>component</b> type, <b>port</b> type, <b>default</b> , <b>template</b> and <b>timer</b> .
<b>testcase</b>	Value, template, port and timer parameterization, type parameterization	Dynamic at run-time	all basic types and of all user-defined types, <b>address</b> type, <b>template</b> .
<b>signature</b>	Value and template parameterization, type parameterization	Dynamic at run-time	all basic types, all user-defined types and <b>address</b> type, <b>component</b> type.

NOTE: Type parameterization is always static at compile-time.

#### 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**.

#### Clause 5.4.1 Formal Parameters

Is extended by the following clause:

##### 5.4.1.5 Formal parameters of kind type

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

#### *Syntactical Structure*

```
[ in ] [ TypeIdentifier ] TypeParIdentifier [ ":" Type ]
```

#### *Semantic Description*

Types 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 parameterization shall be resolved statically.

Type parameters will be written in a separate parameterlist enclosed in angle brackets.

Parameters of type kind may have a default type, which is given by a type assigned to the parameter.

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

### **Restrictions**

- a) Formal type parameters shall be in parameters, which can optionally be indicated by the optional keyword **in**.
- b) The default type has to be compatible with the type of the parameter. For type compatibility see [1] TTCN-3 Core Language clause 6.4. The default type shall not refer to other type parameters in the same parameter list.
- c) Requiring type compatibility of the actual parameter with the formal parameter is possible for component types only.
- d) External functions shall not have type parameters.

### **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;
  :
};
```

#### 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:

### 5.4.2.1 Actual type parameters

Modify the text as follows:

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

#### Syntactical Structure

```
( Expression | _____ // for value parameter
  TemplateInstance | _____ // for template parameter
  TimerRef | _____ // for timer parameter
  Port | _____ // for port parameter
  "-" ) | _____ // to skip a parameter with default
ParameterId " := " ( Expression | TemplateInstance | TimerRef | Port ) )

Type |
TypeParIdentifier " := " Type
```

#### Semantic Description

Is modified as follows:

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

An empty type 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 there shall be an actual parameter. The actual parameter of If a formal parameter with a default is followed by a formal parameter without a default, the actual parameter value can be skipped by using dash "-" as actual parameter. An actual parameter can also be skipped by just leaving it out if no other actual parameter follows in the actual parameter list—either because the parameter is last or because all following formal parameters have default values and are left out If a formal parameter with a default is not followed by a parameter without a default, then the actual parameter can simply be omitted.
- c) When using assignment notation, each formal parameter shall be assigned an actual parameter at most once. For each formal parameter without default value type, there shall be an actual parameter. ~~In order~~ To use the default value type of a formal parameter, no assignment for this specific parameter shall be provided.
- d) The type of each actual parameter shall be compatible with the type of each corresponding formal parameter. 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 ES 201 873-1 [1], clause 6.3.3.
- e) Actual parameters passed to restricted formal template parameters shall obey the restrictions given in clause 15.8 Instantiating formal type parameters with actual types shall result in valid TTCN-3.

EXAMPLES: 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

## 5.3 Extension to 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 parameters.

### Clause 6 Types and values

Is extended by the following 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 [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
};
```

## 5.4 Extension to 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 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

1. FormalTypeParList ::= "<" [FormalTypePar](#) { ", " [FormalTypePar](#) } ">"
2. FormalTypePar ::= [ [InParKeyword](#) ] [ [Type](#) | [TypeKeyword](#) ] [TypeParIdentifier](#) [ [":="](#) [Type](#) ]
3. TypeParIdentifier ::= [Identifier](#)
4. TypeActualParIdentifier ::= [Identifier](#)
5. TypeParAssignment ::= [TypeActualParIdentifier](#) [":="](#) [TypeActualPar](#)
6. ActualTypeParList ::= ( "<" [ActualTypePar](#) { ", " [ActualTypePar](#) } ">" ) |  
 ("<" [ActualTypeParAssignment](#) { ", " [ActualTypeParAssignment](#) } ">" )
7. ActualTypePar ::= [Type](#) | [Dash](#)
8. ActualTypeParAssignment ::= [TypeActualParIdentifier](#) [":="](#) [ActualTypePar](#)
9. StructDefFormalParList ::= (" ([StructDefFormalPar](#) { ", " [StructDefFormalPar](#) } ") )
10. StructDefFormalPar ::= [FormalValuePar](#)

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

18. StructDefBody ::= ([StructTypeIdentifier](#) | [AddressKeyword](#))  
 [[FormalTypeParList](#)] [[StructDefFormalParList](#)]  
 "{" [[StructFieldDef](#) { ", " [StructFieldDef](#) } ] "
34. UnionDefBody ::= ([StructTypeIdentifier](#) | [AddressKeyword](#))  
 [[FormalTypeParList](#)] [[StructDefFormalParList](#)]  
 "{" [[UnionFieldDef](#) { ", " [UnionFieldDef](#) } } "
40. StructOfDefBody ::= ([Type](#) | [NestedTypeDef](#)) ([StructTypeIdentifier](#) | [AddressKeyword](#))  
 [[FormalTypeParList](#)] [[StructDefFormalParList](#)] [[SubTypeSpec](#)]
42. EnumDef ::= [EnumKeyword](#) ([EnumTypeIdentifier](#) | [AddressKeyword](#)) [[StructDefFormalParList](#)]  
 "{" [[EnumerationList](#) ] "
48. SubTypeDef ::= [Type](#) ([SubTypeIdentifier](#) | [AddressKeyword](#))  
 [[FormalTypeParList](#)] [[StructDefFormalParList](#)] [[ArrayDef](#)] [[SubTypeSpec](#)]
56. PortType ::= [[GlobalModuleId](#) [Dot](#)] [PortTypeIdentifier](#) [[ActualTypeParList](#)] [ [TypeActualParList](#) ]
58. PortDefBody ::= [PortTypeIdentifier](#) [[FormalTypeParList](#)] [[StructDefFormalParList](#)] [PortDefAttribs](#)
79. ComponentDef ::= [ComponentKeyword](#) [ComponentTypeIdentifier](#)  
 [[FormalTypeParList](#)] [[StructDefFormalParList](#)]  
 [[ExtendsKeyword](#) [ComponentType](#) { ", " [ComponentType](#) } ]  
 "{" [[ComponentDefList](#) ] "
82. ComponentType ::= [[GlobalModuleId](#) [Dot](#)] [ComponentTypeIdentifier](#)  
 [[ActualTypeParList](#)] [ [TypeActualParList](#) ]
95. BaseTemplate ::= ([Type](#) | [Signature](#)) [TemplateIdentifier](#)  
 [[FormalTypeParList](#)] [" (" [TemplateFormalParList](#) ")"]
107. StructFieldRef ::= [StructFieldIdentifier](#) | [PredefinedType](#) | [ReferencedType](#)  
 /\* **STATIC SEMANTICS** - [PredefinedType](#) and [ReferencedType](#) shall be used for anytype value notation  
 only. [PredefinedType](#) shall not be [AnyTypeKeyword](#).\*/
160. FunctionDef ::= [FunctionKeyword](#) [FunctionIdentifier](#) [[FormalTypeParList](#)]  
 [" ([FunctionFormalParList](#) ") ] [[RunsOnSpec](#)] [[ReturnType](#)] [StatementBlock](#)
177. FunctionInstance ::= [FunctionRef](#) [[ActualTypeParList](#)] [" (" [[FunctionActualParList](#) ] ")"]
182. SignatureDef ::= [SignatureKeyword](#) [SignatureIdentifier](#) [[FormalTypeParList](#)]  
 [" ([SignatureFormalParList](#) ") ] [[ReturnType](#) | [NoBlockKeyword](#)]  
 [[ExceptionSpec](#)]
191. Signature ::= [[GlobalModuleId](#) [Dot](#)] [SignatureIdentifier](#) [[ActualTypeParList](#)]
192. TestcaseDef ::= [TestcaseKeyword](#) [TestcaseIdentifier](#) [[FormalTypeParList](#)]  
 [" ([TestcaseFormalParList](#) ") ] [ConfigSpec](#)  
[StatementBlock](#)
200. TestcaseInstance ::= [ExecuteKeyword](#) [" ([TestcaseRef](#) [[ActualTypeParList](#)]  
 [" (" [[TestcaseActualParList](#) ] ")"]  
 [ ", " [TimerValue](#) ] ")"]
205. AltstepDef ::= [AltstepKeyword](#) [AltstepIdentifier](#) [[FormalTypeParList](#)]  
 [" ([AltstepFormalParList](#) ") ] [[RunsOnSpec](#)]  
 [" {" [AltstepLocalDefList](#) [AltGuardList](#) } "]
211. AltstepInstance ::= [AltstepRef](#) [[ActualTypeParList](#)] [" (" [[FunctionActualParList](#) ] ")"]
456. ReferencedType ::= [[GlobalModuleId](#) [Dot](#)] [TypeReference](#) [[TypeActualParList](#)]  
 [[ExtendedFieldReference](#)]
457. TypeReference ::= [StructTypeIdentifier](#) |  
[EnumTypeIdentifier](#) |  
[SubTypeIdentifier](#) |  
[ComponentTypeIdentifier](#) |  
[PortTypeIdentifier](#)
458. TypeActualParList ::= ( [" (" [TypeActualPar](#) { ", " [TypeActualPar](#) } ")"] ) |  
 ( [" (" [TypeParAssignment](#) { ", " [TypeParAssignment](#) } ")"] )

```

459. TypeActualPar ::= ConstantExpression | TypeActualParIdentifier
603. ExtendedFieldReference ::= {((Dot (StructFieldIdentifier |
                                (TypeDefIdentifier [ActualTypeParList] [TypeActualParList])))
                                | ArrayOrBitRef ) }+
/* STATIC SEMANTIC - The TypeDefIdentifier shall be used only if the type of the VarInstance or
ReferencedValue in wich the ExtendedFieldReference is used is anytype.*/

```

## 6 Package semantics

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 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 data aspects of TTCN 3. This includes aspects like encoding, decoding and the usage of data imported from non TTCN 3 specifications~~. 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 ~~the grouping mechanism~~ value parameterization of types. ~~Grouping is related to The instantiation of value parameters of types has to be done in the definitions part of a TTCN-3 module and has no behavioural aspects. The operational semantics is defined for the instantiated type definitions only.~~

## 7 TRI extensions for the package

Advanced parameterization has no effects on the TRI.

## 8 TCI extensions for the package

NOTE: The TCI logging interface has not yet been extended in this package. It may be done if requested.

### 8.1 Extension to 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`:

<code>TciParameterListType</code> <code>getValueParameters()</code>	Returns a list of the actual value parameters of this type. The list can be empty.
<code>TciParameterTypeListType</code> <code>getTypeParameters()</code>	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 7.3.1.1.7 `tciStartTestCase`

The usage of `tciStartTestCase` is additionally constrained by:

`tciStartTestCase` shall not be used for testcases with type parameters.

## 8.2 Extension to 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.

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

### Clause 9.2 Value interfaces

The interface of Type is extended by the two additional operations.

TCI IDL Interface	ANSI C representation	Notes and comments
<b>Type</b>		
TciParameterListType getValueParameters()	TciParameterListType tciGetValueParameters(Type inst)	
TciParameterTypeListType getTypeParameters()	TciParameterTypeListType tciGetTypeParameters(Type inst)	

### 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

## 8.4 Extension to 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.

```

## 8.5 Extension to 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 ();
};

```

---

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

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

#### Clause 10 Parameterization in ASN.1

Is extended by:

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

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

All parameterized ASN.1 definitions shall be resolvable at compile-time.

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 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 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 @param row 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	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		
@config											X					
@desc	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
@exception								X							X	
@member		X (see note)	X	X	X (see note)	X (see note)	X (see note)								X	
@param	✗	✗	✗	✗	✗	✗	X	X	X	X	X				X	
@purpose											X					
@remark	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
@return								X	X						X	
@see	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		
@status	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
@verdict									X	X	X					
@version	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

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

---

## History

<b>Document history</b>		
V1.0.0	May 2009	Membership Approval Procedure    MV 20090724: 2009-05-26 to 2009-07-24
V1.1.1	July 2009	Publication