

ETSI ES 203 790 V1.1.1 (2019-01)



**Methods for Testing and Specification (MTS);  
The Testing and Test Control Notation version 3;  
TTCN-3 Language Extensions: Object-Oriented Features**

---

**Reference**DES/MTS-203790-00F\_ed111

---

**Keywords**language, TTCN-3

---

**ETSI**

---

650 Route des Lucioles  
F-06921 Sophia Antipolis Cedex - FRANCE

---

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

Siret N° 348 623 562 00017 - NAF 742 C  
Association à but non lucratif enregistrée à la  
Sous-Préfecture de Grasse (06) N° 7803/88

---

**Important notice**

---

The present document can be downloaded from:

<http://www.etsi.org/standards-search>

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

Users of the present document should be aware that the document may be subject to revision or change of status.

Information on the current status of this and other ETSI documents is available at

<https://portal.etsi.org/TB/ETSIDeliverableStatus.aspx>

If you find errors in the present document, please send your comment to one of the following services:

<https://portal.etsi.org/People/CommiteeSupportStaff.aspx>

---

**Copyright Notification**

---

No part may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm except as authorized by written permission of ETSI.

The content of the PDF version shall not be modified without the written authorization of ETSI.

The copyright and the foregoing restriction extend to reproduction in all media.

© ETSI 2019.

All rights reserved.

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

**3GPP™** and **LTE™** are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners.

**oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners.

**GSM®** and the GSM logo are trademarks registered and owned by the GSM Association.

# Contents

Intellectual Property Rights .....	5
Foreword.....	5
Modal verbs terminology.....	5
1 Scope .....	6
2 References .....	6
2.1 Normative references .....	6
2.2 Informative references.....	6
3 Definition of terms and abbreviations .....	7
3.1 Terms.....	7
3.2 Abbreviations .....	7
4 Package conformance and compatibility.....	7
5 Package Concepts for the Core Language.....	8
5.0 General .....	8
5.1 Classes and Objects .....	8
5.1.0 General.....	8
5.1.1 Classes .....	8
5.1.1.0 General .....	8
5.1.1.1 Scope rules .....	9
5.1.1.2 Abstract classes .....	10
5.1.1.3 External classes .....	10
5.1.1.4 Final Classes .....	10
5.1.1.5 Constructors .....	11
5.1.1.6 Destructors .....	11
5.1.1.7 Methods.....	11
5.1.1.8 Method invocation .....	12
5.1.1.9 Visibility .....	12
5.1.1.10 Built-in classes .....	12
5.1.2 Objects .....	13
5.1.2.0 General .....	13
5.1.2.1 Ownership .....	13
5.1.2.2 Object References .....	13
5.1.2.3 Null reference.....	13
5.1.2.4 Select class-statement.....	13
5.1.2.5 Of-operator (Dynamic Class Discrimination) .....	14
5.1.2.6 Casting .....	14
5.2 Exception handling.....	14
5.2.0 General.....	14
5.2.1 Extension to ETSI ES 201 873-1, clause 16.1.0 (Functions).....	14
5.2.2 Extension to ETSI ES 201 873-1, clause 16.1.3 (External Functions) .....	15
5.2.3 Extension to ETSI ES 201 873-1, clause 16.1.4 (Invoking functions from specific places).....	15
5.2.4 Extension to ETSI ES 201 873-1, clause 16.2 (Altsteps).....	15
5.2.5 Extension to ETSI ES 201 873-1, clause 16.3 (Test cases) .....	16
5.2.6 Extension to ETSI ES 201 873-1, clause 18 (Overview of program statements and operations) .....	16
5.2.7 Extension to ETSI ES 201 873-1, clause 19 (Basic program statements) .....	18
6 TRI Extensions for the Package .....	21
6.1 Extensions to clause 5.3 of ETSI ES 201 873-5 Data interface.....	21
6.2 Extensions to clause 5.6.3 of ETSI ES 201 873-5 Miscellaneous operations .....	22
6.3 Extensions to clause 6 of ETSI ES 201 873-5 Java™ language mapping.....	24
6.4 Extensions to clause 7 of ETSI ES 201 873-5 ANSI C language mapping.....	25
6.5 Extensions to clause 8 of ETSI ES 201 873-5 C++ language mapping.....	26
6.6 Extensions to clause 9 of ETSI ES 201 873-5 C# language mapping .....	27
7 TCI Extensions for the Package .....	28

7.1	Extensions to clause 7.2.2.1 of ETSI ES 201 873-6 Abstract TTCN-3 data types and values .....	28
7.2	Extensions to clause 7.2.2 of ETSI ES 201 873-6 Abstract TTCN-3 data types and values .....	28
7.3	Extensions to clause 7.2.2.2.0 of ETSI ES 201 873-6 Basic rules .....	30
7.4	Extensions to clause 7.2.2.2 of ETSI ES 201 873-6 Abstract TTCN-3 values .....	30
7.5	Extensions to clause 7.3.4.1 of ETSI ES 201 873-6 Abstract TCI-TL provided .....	31
7.6	Extensions to clause 8 of ETSI ES 201 873-6 Java™ language mapping .....	33
7.7	Extensions to clause 9 of ETSI ES 201 873-6 ANSI C language mapping .....	35
7.8	Extensions to clause 10 of ETSI ES 201 873-6 C++ language mapping .....	37
7.9	Extensions to clause 11 of ETSI ES 201 873-6 W3C XML mapping .....	39
7.10	Extensions to clause 12 of ETSI ES 201 873-6 C# language mapping .....	40
8	<b>XTRI Extensions for the Package (optional) .....</b>	<b>42</b>
8.1	Changes to clause 5.6.3 of ETSI ES 201 873-5 Miscellaneous operations .....	42
8.2	Extensions to clause 6 of ETSI ES 201 873-5 Java™ language mapping .....	44
8.3	Extensions to clause 7 of ETSI ES 201 873-5 ANSI C language mapping .....	44
8.4	Extensions to clause 8 of ETSI ES 201 873-5 C++ language mapping .....	45
8.5	Extensions to clause 9 of ETSI ES 201 873-5 C# language mapping .....	45
	<b>Annex A (normative): BNF and static semantics .....</b>	<b>46</b>
A.1	Extensions to TTCN-3 terminals .....	46
A.2	Modified TTCN-3 syntax BNF productions .....	47
A.3	Additional TTCN-3 syntax BNF productions .....	48
	History .....	49

---

## 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 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 support for object-oriented features in 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 OMG 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.

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)".

### 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.

- [i.1] 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".

- [i.2] 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".
- [i.3] 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".
- [i.4] 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".

---

## 3 Definition of terms 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] and ETSI ES 201 873-6 [4] apply.

### 3.2 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] and ETSI ES 201 873-6 [4] apply.

---

## 4 Package conformance and compatibility

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

"TTCN-3:2018 Object-Oriented features" - 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 ETSI ES 201 873-1 [1] and ETSI ES 201 873-4 [2].

The package presented in the present document is compatible to:

- ETSI ES 201 873-1 [1], version 4.10.1;
- ETSI ES 201 873-4 [2], version 4.6.1;
- ETSI ES 201 873-5 [3], version 4.8.1;
- ETSI ES 201 873-6 [4], version 4.9.1;
- ETSI ES 201 873-7 [i.1];
- ETSI ES 201 873-8 [i.2];
- ETSI ES 201 873-9 [i.3];
- ETSI ES 201 873-10 [i.4].

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

### 5.0 General

This package defines objec-oriented features for TTCN-3, i.e. it extends the TTCN-3 core language (ETSI ES 201 873-1 [1]) with well-known concepts from object-oriented programming and modelling languages. This package realizes the following concepts:

- classes (i.e. class definition, scope rules, abstract and external classes, refinement, constructors, destructors, methods, visibility, and built-in classes);
- objects (i.e. ownership, object references, select class-statement, dynamic class discrimination, and casting); and
- exception handling (i.e. ability to define exception handling for functions, external functions, altsteps, and test cases).

### 5.1 Classes and Objects

#### 5.1.0 General

This clause introduces the concepts of class types and their values, called objects as well as the operations allowed to be applied to these objects.

#### 5.1.1 Classes

##### 5.1.1.0 General

###### *Syntactical Structure*

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

###### *Semantic Description*

A class is a type where the values are called objects. A class can declare fields (variables, constants, templates, ports, timers) and methods as its members. Each member name inside the class shall be unique, there is no overloading. The private and protected fields and methods are only accessible by the methods of the class, while the public members of the class can be accessed also from behaviour not defined in the class. The private members of the class can be accessed directly only by members of the class itself. All members which are neither private nor public are protected and can also be accessed by members of subclasses.

A class can extend another class. The extended class is called the superclass, while the extending class is called the subclass. The resulting type of a class definition is the set of object instances of the class itself and all instances of its direct or indirect subclasses. A subclass is a subtype of its direct and indirect superclasses and its object instances are type compatible with them. If a class does not explicitly extend another class type, it implicitly extends the root class type object. Thus, all classes are directly or indirectly extensions of the object class.

A class can have optional runs on, mtc and system clauses. This restricts the type of component context that can create objects of that class and all methods of this class. If a class does not have one of these clauses, it inherits it from its superclass, if the superclass has one. If the superclass has or inherits a runs on, mtc or system clause, the subclass may declare each of these clauses with a more specific component type than the one inherited. The function members of classes shall not have runs on, system or mtc classes but inherit them from their surrounding class or its superclasses.



**Restrictions**

- a) Templates are not allowed for class types.
- b) Passing of object references to the create operation of a component type or a function started on another component is not allowed.
- c) No subtyping definition is allowed for class types via the normal subtype definition.
- d) No local/global constants or module parameters of class type or containing class type fields or elements are allowed.
- e) Class type cannot be the contained value of an anytype value.
- f) The functions of a class shall not have a runs on, mtc or system clause.
- g) The runs on type of a class shall be runs on compatible with the runs on type of the behaviour creating a class.
- h) The runs on type of a class shall be runs on compatible with the runs on type of the superclass.
- i) The mtc and system type of a class shall be mtc and system compatible with the mtc and system types of the superclass, respectively.

**5.1.1.1 Scope rules**

Class constitutes a scope unit. For the uniqueness of identifiers, the rules specified in the clause 5.2.2 of ETSI ES 201 873-1 [1] apply with the following exceptions:

- a) Identifiers from the higher scope can be reused for member declarations. A reference to a reused identifier without a prefix occurring inside a class scope shall be resolved as a reference to the class member. In order to refer to the declaration on the higher scope, the identifier shall be preceded with a module name and a dot (".").
- b) Identifiers of member declarations can be reused inside methods for formal parameter and local declarations. A reference to a reused identifier without a prefix occurring inside a class method shall be resolved as a reference to the formal parameter or local declaration. In order to refer to the member declaration, the identifier shall be preceded with the `this` keyword and a dot.
- c) Reusing identifiers of members of the component type specified in the runs on clause of the class for members and inside methods for formal parameters and local declarations is not allowed.

EXAMPLE:

```

module ClassModule {
  const integer a := 1;

  type class MyClass() {
    const integer a := 2;
    function doSomething (integer a := 3) {
      log(a); // logs 3 (for the default value)
      log(this.a); // logs 2
      log(ClassModule.a); // logs 1
    }
    function doSomethingElse () {
      log(a); // logs 2
      log(this.a); // also logs 2
      log(ClassModule.a); // logs 1
    }
  }
}

```

### 5.1.1.2 Abstract classes

A class can be declared as `@abstract`. In that case, it is allowed that it also declares abstract member functions who shall be defined by all non-abstract subclasses. An abstract method function has no function body but can be called in all concrete instances of subclasses of the abstract class declaring it. Other members of the abstract class or its subclasses may use the abstract functions as if it was concrete where at runtime the concrete overriding definition will be used.

NOTE 1: Abstract classes are only useful as superclasses of concrete classes.

#### **Restrictions**

- a) Abstract classes cannot be explicitly instantiated.
- b) If a class that is not declared abstract extends an abstract class, all methods that have no implementation in the superclass shall be implemented in this class.

NOTE 2: Variables of an abstract class type can only contain references to instances of non-abstract subclasses.

### 5.1.1.3 External classes

A class may also be declared as external. In that case, all members shall be external functions without a function body. It is allowed to omit the external keyword from these function declarations. When instantiating an external class, the object being created is provided by the platform adapter and the method calls to the object are delegated via the platform adapter to the corresponding method of the external object.

NOTE 1: External classes are a way to use object-oriented library functionality to TTCN-3 while still remaining abstract and independent of actual implementation. Libraries for common constructs like stacks, collections, tables can be defined or automatic import mechanisms could be provided.

If an object of an external class is instantiated, it implicitly creates an external object and the internal object has a handle to the external one. The reference to the external object is called a handle. When an external method is invoked on the internal object, the call is delegated to the handle.

NOTE 2: External objects are possibly shared between different parts of the test system. Therefore, racing conditions and deadlocks have to be avoided by the external implementation.

An internal class can extend an external class and add internal behaviour, but also additional external functions, which have to be declared external explicitly. Such a class is conceptually still an external class and each instance has a handle to an external object.

#### **Restrictions**

- a) External classes shall not contain fields or functions with a body.
- b) External classes shall not be derived from non-external classes other than object.
- c) If an internal class defines an external function, it shall be derived from an external class either directly or indirectly.

EXAMPLE:

```
external type class Stack {
  function push(integer v);
  function pop() return integer;
  function isEmpty() return boolean;
}
```

### 5.1.1.4 Final Classes

If a class shall not be subclassed, it may be declared as `@final`. Final classes cannot be abstract.

### 5.1.1.5 Constructors

#### *Syntactic Structure*

```

create "(" { FormalParameter , }* ")"
[":" SuperClass "(" { ActualParameter , }+ ")" ]
StatementBlock

```

#### *Semantic Description*

A class can define a constructor called `create`. If no constructor is defined, a default constructor is implicitly provided where the formal parameters of the constructor are the parameters of the (implicit or explicit) constructor of the direct superclass and one formal in parameter for each declared member field of the class itself in their order of declaration with equivalent type. The constructor is invoked on a type reference to the class and the result of this invocation is a new instance object of the constructor's specific class. If a class is extending another class with an explicit constructor, that constructor shall be invoked by adding a super-constructor clause with an actual parameter list to the constructor declaration. An implicit constructor will automatically pass the required actual parameters to the constructor of its superclass.

In the constructor, it is allowed to refer to the object being constructed as `this` to reference the fields of the object to be created in case that the names of the formal parameters clash with the names of those fields. They are explicitly allowed to have the same names as class members.

#### EXAMPLE:

```

type class MyClass {
  var integer a;
  const float b;
  // implicit constructor:
  // create(integer a, float b) {
  //   this.a := a;
  //   this.b := b
  // }
}

type class MyClass2 extends MyClass {
  template integer t;
  // explicit constructor
  create(template integer t) : MyClass(2, 0.5) {
    this.t := t;
  }
}

```

### 5.1.1.6 Destructors

#### *Syntactic Structure*

```

finally StatementBlock

```

#### *Semantic Description*

A destructor may be provided using a `finally` declaration following the class body. This destructor will be invoked automatically at the latest before the system deallocates an object instance (which is tool specific and out of the scope of the present document) or when the owning component is terminates. The *StatementBlock* has access to all members accessible to the class. The *StatementBlock* is semantically a function body of a function without return clause.

When deallocating the object instance, the destructor of the associated class is invoked first, followed by the destructor of all parent classes in the reverse order of superclass hierarchy.

### 5.1.1.7 Methods

A method is a function defined inside the class body. It has the same properties and restrictions as any normal function, but it is invoked in an object which can be referred to by the `this` object reference. A method invocation can access the class's own fields and also the inherited protected fields and methods of its superclasses.

A method inherited from a superclass can be overridden by the subclass by redefining a function of the same name and with the same formal parameter list. When a method is called in an object, the version of the most specific class of the super class hierarchy of the concrete class that defines the method in its body will be invoked. The overridden method can be invoked from the overriding class by using the keyword `super` as the object reference of the invocation. If a method shall not be overridden by any subclass, it can be declared as `@final`.

Public methods, if not overridden by the subclass, are inherited from the superclasses. If a public method is declared in a class, it can be invoked also in all objects of its direct or indirect subclasses.

If a public method is overridden, the overriding method shall have the same formal parameters in the same order as the overridden method. Public methods shall be overridden only by public methods. Protected methods may be overridden by public or protected methods.

The return type of an overriding function shall be the same as the return type of the overridden function with the same template restrictions and modifiers.

Methods shall have no `runs on`, `system` or `mtc` clause directly attached to them. However, they inherit these clauses from their surrounding class.

### 5.1.1.8 Method invocation

#### *Syntactical Structure*

```
[ObjectInstance "."] Identifier "(" FunctionActualParList ")"
```

A method invocation is a function call associated with a certain object defined in the class of that object.

Methods are invoked using the dotted notation on an object reference. Inside the scope of a class, methods of the same class or any visible inherited methods can be invoked without the *ObjectInstance* prefix if the object the method shall be invoked in is the same object as the one invoking it. The usual restrictions on actual parameters, as well as `runs on`, `mtc` and `system` types apply also on method invocations. All other restrictions that apply to called functions also apply to method invocation.

### 5.1.1.9 Visibility

Fields can be declared as `private` or `protected`. Methods can be declared as `private`, `public` or `protected`. If no visibility is given then the default modifier `protected` is assumed.

Private member functions are not visible and can be present in multiple classes of the same hierarchy with different parameter lists and return values.

Public member functions can be called from any behaviour running on the object's owner component.

#### *Restrictions*

- a) A field of any visibility cannot be overridden by a subclass.
- b) A public member function can only be overridden by another public member function.
- c) Private members can only be accessed directly from inside their surrounding class's scope.

### 5.1.1.10 Built-in classes

The abstract special built-in class called `object` is the superclass for all classes that do not explicitly extend another class.

The pseudo definition of that class is:

```
type class @abstract @builtin object {
    // This function will return a tool-specific descriptive string by default
    // but can be overridden by subclasses
    public function toString() return universal charstring;
}
```

NOTE: The `@builtin` is only added for illustrative purposes and not part of the TTCN-3 language.

## 5.1.2 Objects

### 5.1.2.0 General

Objects are the instances of classes. Each instance comprises an instance of the data of the fields of the class (including all superclasses) and allows invocation of its public methods by other behaviour and protected or private methods by behaviour defined by the object's class itself.

### 5.1.2.1 Ownership

Each object is owned by the component on which it was created. The owning component of an object can be referenced via the `self` component reference. Methods of objects can only be invoked by behaviour that also runs on the owning component. An object is created on a component if its constructor was invoked by a behaviour running on that component.

### 5.1.2.2 Object References

Objects are always passed by reference (even though their formal parameters can still be `in`, `inout` or `out`, dependent on the usage of that parameter). A variable of a class type contains only a reference to the object instance and the object is not copied when used as an actual parameter or assigned to a variable, but only the reference to the object. Therefore, multiple variables can contain a reference to the same object simultaneously.

#### **Restrictions**

- a) Object References shall not be passed as actual parameter or part of an actual parameter to either the create operation of a component type or a function started on a component. If a structured type contains a field of a class type, this type is not seen as a data type and its values cannot be used for sending and receiving or as an argument to any expression other than the equality/inequality operator.

NOTE: Since objects cannot be shared by different component contexts and for each component at most one behaviour is running, no parallel conflicting access to any of the objects fields or methods is possible.

### 5.1.2.3 Null reference

An object variable that is not initialized with an object instance contains the special value `null`. An object variable or parameter may be compared with the special value `null` with the equality and inequality operators or can be assigned the special value `null` explicitly.

### 5.1.2.4 Select class-statement

#### **Syntactical Structure**

```
select class "(" Object ")"
"{ " { case "(" ClassReference ")" StatementBlock }+ [ElseCase] " }
```

#### **Semantic Description**

The class of an object can be discriminated for via the 'select class' statement that is similar to a select union statement insofar that it allows only superclasses and known subclasses of the object reference's class in the context. If more than one case contains a superclass of the actual class of the given object instance, the first of these cases will be chosen by the select class statement.

#### EXAMPLE:

```
type class A {}
type class B extends A {}
...
var A v_a := B.create();
select class (v_a) {
  case (B) { ... } // will be chosen
  case (A) { ... } // will not be chosen
}
```

**Restrictions**

- a) If a class from one case is a superclass of a class from another case, then the case of the subclass shall be precede the case of the superclass.

**5.1.2.5 Of-operator (Dynamic Class Discrimination)****Syntactical Structure**

Object **of** ClassReference

**Semantic Description**

To check whether an object is an instance is of a certain class, the **of** operator may be used.

It yields a Boolean value which is true if and only if the most specific class of the object referenced on the left-hand side is either equal to or a subclass derived from the class type reference on the right-hand side.

**5.1.2.6 Casting****Syntactical Structure**

ObjectReference **"=>"** ( ClassIdentifier | " (" ClassReference ")" )

**Semantic Description**

An object reference can be cast to another class of the object's known class's set of direct or indirect superclasses and direct or indirect subclasses. This operation yields an object reference to the same object but can be used as being of the type being cast to. If the referenced class to be cast to is an expression that is not a simple identifier, the expression shall be written in parenthesis.

**Restrictions**

- a) If the class the object is being cast to is not in the set of superclasses or the concrete class of the object, the cast operation shall result in an error.

**5.2 Exception handling****5.2.0 General**

This clause introduces exception handling into TTCN-3. It provides means to define exception handling for functions, external functions, altsteps, and test cases.

**5.2.1 Extension to ETSI ES 201 873-1, clause 16.1.0 (Functions)****Clause 16.1.0 General**

The syntax of functions is extended with an optional **exception** clause.

**Syntactical Structure**

```
function [ @deterministic ] FunctionIdentifier
"(" { ( FormalValuePar | FormalTimerPar | FormalTemplatePar | FormalPortPar ) [","] } ")"
[ runs on ComponentType ]
[ mtc ComponentType ]
[ system ComponentType ]
[ return [ template ] Type ]
[ exception "(" {Type [","]}+ ")" ]
StatementBlock
```

### Clause 16.1.0 General

The semantic description part is extended.

Functions may have an exception list. The exception list declares, what exception types may be raised during the execution of the function either directly or indirectly.

NOTE 1: The exception list can be used to communicate to the callers of the function what exceptions to prepare for and by tools to perform stronger static checks. For backward compatibility reasons the exception list is optional.

NOTE 2: The exception list might not be exhaustive. With activated altsteps it might not be possible to precisely know what exceptions might be raised within a function directly or indirectly.

If the statement block of a function has a **finally** block, the finally block is always executed before control returns to the location of the call of the function.

## 5.2.2 Extension to ETSI ES 201 873-1, clause 16.1.3 (External Functions)

### Clause 16.1.3 General

The syntax of external functions is extended with the optional **exception** clause.

#### *Syntactical Structure*

```
external function [ @deterministic ] ExtFunctionIdentifier
 "(" { ( FormalValuePar | FormalTimerPar | FormalTemplatePar | FormalPortPar ) [ "," ] } ")"
 [ return [ template [ Restriction ] ] Type ] [ exception "(" {Type [ "," ]}+ ")" ]
```

### Clause 16.1.3 General

The semantic description part is extended.

External functions may have an exception list. The exception list declares, what exception types may be raised during the execution of the external function.

NOTE 0: The exception list can be used by tools to perform stronger static checks. For backward compatibility reasons the exception list is optional.

NOTE 1: The exception list might not be exhaustive. It might not be possible to precisely know what exceptions might be raised within an external function directly or indirectly.

## 5.2.3 Extension to ETSI ES 201 873-1, clause 16.1.4 (Invoking functions from specific places)

### Clause 16.1.4 General

The list of restrictions is extended to avoid side effects.

- m) Raising an exception with the raise exception statement.

## 5.2.4 Extension to ETSI ES 201 873-1, clause 16.2 (Altsteps)

### Clause 16.2.0 General

The syntax of altstep is extended with the optional **exception**, **catch** and **finally** clauses.

#### *Syntactical Structure*

```
altstep AltstepIdentifier
 "(" { ( FormalValuePar | FormalTimerPar | FormalTemplatePar | FormalPortPar ) [ "," ] } ")"
 [ runs on ComponentType ]
 [ mtc ComponentType ]
 [ system ComponentType ]
 [ exception "(" {Type [ "," ]}+ ")" ]
 "{ "
```

```

    { ( VarInstance | TimerInstance | ConstDef | TemplateDef ) [ ";" ] }
    AltGuardList
  }"
  { CatchBlock }
  [ FinallyBlock ]

```

### Clause 16.2.0 General

The semantic description part is extended.

Altsteps may have an exception list. The exception list declares, what exception types may be raised during the execution of the altstep either directly or indirectly.

NOTE 0: The exception list can be used to communicate to the callers of the altstep what exceptions to prepare for and by tools to perform stronger static checks. For backward compatibility reasons the exception list is optional.

NOTE 1: The exception list might not be exhaustive. With activated altsteps it might not be possible to precisely know what exceptions might be raised within an altstep directly or indirectly.

Altsteps may have a finally block. If present the finally block is always executed before control returns to the location of the call of the altstep every time the altstep is invoked, regardless of whether implicitly as an activated default or explicitly from another alt statement.

## 5.2.5 Extension to ETSI ES 201 873-1, clause 16.3 (Test cases)

### Clause 16.3 Test cases

The semantic description part is extended.

The StatementBlock of Test cases may have a finally block. If present the finally block is always executed before the test case terminates.

Exceptions raised directly or indirectly within the test case and not handled latest by the catch clauses of the StatementBlock of the testcase results in the testcase finishing with a dynamic error. In this situation the dynamic error has to reference not handling the exception as the reason of error.

NOTE 0: The reason for the dynamic error is not the raising of the exception, but the lack of handling within the testcase.

## 5.2.6 Extension to ETSI ES 201 873-1, clause 18 (Overview of program statements and operations)

The list of statements in table 15 of ETSI ES 201 873-1 needs to be extended with a raise exception statement as shown below.

Statement	Associated keyword or symbol	Can be directly or indirectly invoked by module control, but not by test components	Can be invoked by functions, test cases and altsteps running on test components	Can be directly or indirectly invoked from specific places (see note 1)
Expressions	(...)	Yes	Yes	Yes
<b>Basic program statements</b>				
Assignments	:=	Yes	Yes	Yes (see note 4)
If-else	if (...) {...} else {...}	Yes	Yes	Yes
Select case	select case (...) { case (...) {...} case else {...}}	Yes	Yes	Yes
For loop	for (...) {...}	Yes	Yes	Yes
While loop	while (...) {...}	Yes	Yes	Yes
Do while loop	do {...} while (...)	Yes	Yes	Yes
Label and Goto	label / goto	Yes	Yes	Yes
Stop execution	stop	Yes	Yes	



Statement	Associated keyword or symbol	Can be directly or indirectly invoked by module control, but not by test components	Can be invoked by functions, test cases and altsteps running on test components	Can be directly or indirectly invoked from specific places (see note 1)
Returning control	<b>return</b>		Yes (see note 5)	Yes
Leaving a loop, alt, altstep or interleave	<b>break</b>	Yes	Yes	Yes
Next iteration of a loop	<b>continue</b>	Yes	Yes	Yes
Raise exception	<b>raise</b>	Yes	Yes	Yes
Logging	<b>log</b>	Yes	Yes	Yes
<b>Statements and operations for alternative behaviours</b>				
Alternative behaviour	<b>alt {...}</b>	Yes (see note 2)	Yes	
Re-evaluation of alternative behaviour	<b>repeat</b>	Yes	Yes	
Interleaved behaviour	<b>interleave {...}</b>	Yes (see note 2)	Yes	
Activate a default	<b>activate</b>	Yes	Yes	
Deactivate a default	<b>deactivate</b>	Yes	Yes	
<b>Configuration operations</b>				
Create parallel test component	<b>create</b>		Yes	
Connect component port to component port	<b>connect</b>		Yes	
Disconnect two component ports	<b>disconnect</b>		Yes	
Map port to test interface	<b>map</b>		Yes	
Unmap port from test system interface	<b>unmap</b>		Yes	
Get MTC component reference value	<b>mtc</b>		Yes	Yes
Get test system interface component reference value	<b>system</b>		Yes	Yes
Get own component reference value	<b>self</b>		Yes	Yes
Start execution of test component behaviour	<b>start</b>		Yes	
Stop execution of test component behaviour	<b>stop</b>		Yes	
Terminating the testcase with an error verdict	<b>testcase.stop</b>		Yes	Yes
Remove a test component from the system	<b>kill</b>		Yes	
Check termination of a PTC behaviour	<b>running</b>		Yes	
Check if a PTC exists in the test system	<b>alive</b>		Yes	
Wait for termination of a PTC behaviour	<b>done</b>		Yes	
Wait a PTC cease to exist	<b>killed</b>		Yes	
<b>Communication operations</b>				
Send message	<b>send</b>		Yes	
Invoke procedure call	<b>call</b>		Yes	
Reply to procedure call from remote entity	<b>reply</b>		Yes	
Raise exception (to an accepted call)	<b>raise</b>		Yes	
Receive message	<b>receive</b>		Yes	
Trigger on message	<b>trigger</b>		Yes	
Accept procedure call from remote entity	<b>getcall</b>		Yes	
Handle response from a previous call	<b>getreply</b>		Yes	
Catch exception (from called entity)	<b>catch</b>		Yes	
Check (current) message/call received	<b>check</b>		Yes	
Clear port queue	<b>clear</b>		Yes	
Clear queue and enable sending & receiving at a to port	<b>start</b>		Yes	
Disable sending and disallow receiving operations to match at a port	<b>stop</b>		Yes	

Statement	Associated keyword or symbol	Can be directly or indirectly invoked by module control, but not by test components	Can be invoked by functions, test cases and altsteps running on test components	Can be directly or indirectly invoked from specific places (see note 1)
Disable sending and disallow receiving operations to match new messages/calls	<b>halt</b>		Yes	
Check the state of a port	<b>checkstate</b>		Yes	
<b>Timer operations</b>				
Start timer	<b>start</b>	Yes	Yes	
Stop timer	<b>stop</b>	Yes	Yes	
Read elapsed time	<b>read</b>	Yes	Yes	
Check if timer running	<b>running</b>	Yes	Yes	
Timeout event	<b>timeout</b>	Yes	Yes	
<b>Verdict operations</b>				
Set local verdict	<b>setverdict</b>		Yes	
Get local verdict	<b>getverdict</b>		Yes	Yes
<b>External actions</b>				
Stimulate an (SUT) action externally	<b>action</b>	Yes	Yes	
<b>Execution of test cases</b>				
Execute test case	<b>execute</b>	Yes	Yes (see note 3)	
NOTE 1: Specific places are defined in clause 16.1.4. Only operations that do not have any potential side effects on snapshot evaluation are allowed.				
NOTE 2: Can be used to control timer operations only.				
NOTE 3: Can only be used in functions and altsteps that are used in module control.				
NOTE 4: Changing of component variables is disallowed.				
NOTE 5: Can be used in functions and altsteps but not in test cases.				

## 5.2.7 Extension to ETSI ES 201 873-1, clause 19 (Basic program statements)

### Clause 19.0 General

The list of statements in table 17 needs to be extended with the raise exception statement as described below.

Basic program statements	
Statement	Associated keyword or symbol
Assignments	<b>:=</b>
If-else	<b>if (...) {...} else {...}</b>
Select case	<b>select case (...) { case (...) {...} case else {...} }</b>
For loop	<b>for (...) {...}</b>
While loop	<b>while (...) {...}</b>
Do while loop	<b>do {...} while (...)</b>
Label and Goto	<b>label / goto</b>
Stop execution	<b>stop</b>
Returning control	<b>return</b>
Leaving a loop, alt, altstep or interleave	<b>break</b>
Next iteration of a loop	<b>continue</b>
Raise exception	<b>raise</b>
Logging	<b>log</b>

### Clause 19.14 Statement Block

The syntax of statement block is changed as shown below.

```
BasicStatementBlock: "{" { LocalDefinition | Statement } }"
StatementBlock: BasicStatementBlock {catch "(" Type Identifier ")" BasicStatementBlock }
[finally BasicStatementBlock]
```

### Clause 19.14 Statement Block

The semantic description part is extended.

A basic statement block is a sequence of declarations and statements.

Statement blocks can be used like basic program statements to introduce a local scope in the flow of control of TTCN-3 behaviour. The declarations and statements in a basic statement block are executed in the order of their appearance, i.e. sequentially.

A statement block consists of a basic statement block with optional additional catch clauses, that can be used to handle exceptions raised directly or indirectly within the basic statement block and an optional finally clause which is executed after the basic statement block execution. When an exception is raised by a statement in the basic statement block the catch clauses are tried in order of appearance to find one of the same type for data types or one the exception can be cast to if it is a type class kind exception. Execution continues with the basic stament block of the first catch clause whose type matches the type of the raised exception.

The catch clause declares a variable of an exception, with the type and identifier provided, to hold the value of the exception within the catch clause. The scope of this variable is limited to the basic statement block of the catch clause, i.e. it is only visible inside the body of the catch clause.

NOTE: The scope of the catch and finally blocks is on the same level with the scope of the basic statement block. Local variables declared within the basic statement block are not visible in the catch and finally clauses.

### Clause 19.14 Statement Block

The list of restrictions is extended

- a) The control transfer statements **return**, and **raise** shall not be used in the **finally** clause. Functions that can raise exceptions shall not be called in the finally clause.
- b) The basic statement block of a **catch** clause shall obey the same semantic restrictions as the basic statement block it follows.

Clause 19 is extended with a new clause.

### NEW: Clause 19.15 The Raise exception statement

The **raise** exception statement raises an exception, causing the execution to continue at the catch block closest in the procedure call hierarchy, also executing all **finally** blocks it encounters while traversing the procedure call hierarchy.

#### *Syntactical Structure*

```
raise TemplateInstance
```

#### *Semantic Description*

The **raise** statement is used to raise an exception. On executing a **raise** exception statement the statement blocks, loops, **alt** statements or **interleave** statement within the encompassing function/altstep/testcase are left. If the encompassing function, altstep or testcase has a catch block with the exact same type as that of the raised exception value for data types or one the exception can be cast to if it is a class type exception, execution continues in that catch block. If the encompassing function or altstep does not have catch blocks or none of the catch blocks can handle the raised exception, execution leaves the function or altstep to handle the exception in the calling function, altstep or testcase. An exception not handled via catch clause of the StatementBlock of a testcase shall cause a dynamic error.

If the StatementBlock of a function, altstep or testcase has a finally block, this finally block is always executed before the function, altstep, testcase terminates. If an exception was raised and handled in a catch block, the finally block is executed after the catch block. If there was no exception raised, or an exception was raised but not handled in any catch blocks the finally block is executed before the function, altstep or testcase terminates.

The parameter of the **raise** operation shall evaluate to a value, that the exception will have.

Exceptions are specified as types. Therefore the exception value may either be derived from a template conforming to the template(value) restriction or be the value resulting from an expression (which of course can be an explicit value). The type of the value specification to the **raise** operation shall be determinable as it is necessary to avoid any ambiguity of the type of the value being raised.

NOTE 0: The type of the raised exception should be provided explicitly for literal values. Catch clauses with synonym types or restricted types will only catch exceptions of the same type.

### Restrictions

In addition to the general static rules of TTCN-3 given in clause 5 and shown in table 15, the following restrictions apply:

- a) An exception shall only be raised inside a function, altstep or testcase.
- b) The *TemplateInstance* shall conform to the template(value) restriction (see clause 15.8).
- c) Exceptions shall not be raised directly or indirectly inside finally blocks of functions, altsteps or testcases.

### Examples

#### EXAMPLE 1:

```
raise ( v_myVariable + v_yourVariable - 2);
// Raises an exception with a value which is the result of the arithmetic expression

raise integer:5;    // Raises an exception with the integer value 5

raise charstring:"Olala!";
// Raises an exception with the charstring value "Olala!"
```

#### EXAMPLE 2: Catching an exception raised in a called function.

```
type record of charstring t_registeredNames;
type component myComponent {
    var t_registerdNames v_registeredNames;
}
function f_init(in charstring name) exception (charstring, integer) runs on myComponent
{
    ...
    if (name_was_not_registered) {
        raise ("Could not initialize " & name); // when the exception is raised f_init teminates
    }
    ...
}

function f_operation(in charstring user1, in charstring user2) exception (integer)
runs on myComponent {
    f_init(user1);
    f_init(user2);
    ...
} catch (charstring e) {
    // the exception is available for processing in the e variable
    // release resources and terminate function
} catch (integer e) {
    //there was some other issue
    // release resources
    raise e; /// the exception is raised again to be handled in the calling function
}
```

#### EXAMPLE 3: Finally is always executed.

```
function f_operation2(in charstring user1, in charstring user2) exception (charstring)
runs on myComponent {
    f_init(user1);
    f_init(user2);
```

```

...
} finally {
// finally is executed wether there was an exception or not before the function terminates
}

```

EXAMPLE 4: The exception can travel through several functions in the call hierarchy until handled.

```

function f_operation3(in charstring user1, in charstring user2) exception (charstring)
runs on myComponent {
  f_operation2(user1, user2); // an exception is raised in f_init
  ...
} finally {
// after the finally block in f_operation2 this finally block is also executed
// the exception is not caught.
}

```

EXAMPLE 5: Exception not caught latest in a testcase is reported as dynamic error.

```

testcase t_myTest1() runs on myComponent {
  f_init("user1");
  f_init("unknown user");// bad argument will raise an exception in f_init
  ... // because of the raised exception execution continues in the finally block
} finally {
  ... // via the runs on component resources can be freed
  // as the exception is not caught dynamic error is reported
}

```

EXAMPLE 6: The type of the exception has to match the type of the catch clause exactly.

```

function f_example() exception (integer) {
  raise integer:5;
}

type integer MyIntegerSynonim;
type integer MyIntegerRange (0 .. 255);

function f_example2() {
  f_example();
} catch (MyIntegerRange e) {
// The exception is not caught here.
// The type of the raised exception and the type of the catch type has to be the same
} catch (MyIntegerSynonim e) {
// The exception is not caught here.
// The type of the raised exception and the type of the catch type has to be the same
} catch (integer e) {
// As the exception raised in f_example was raise with the integer type it is handled here
}

```

## 6 TRI Extensions for the Package

### 6.1 Extensions to clause 5.3 of ETSI ES 201 873-5 Data interface

#### Clause 5.3.2 Communication

The clause is to be modified:

TriExceptionType

A value of type `TriExceptionType` is an encoded type and value of an exception that either is to be sent to the SUT or has been received from the SUT. This abstract type is used in procedure based TRI communication operations and raising exception during execution of external functions, constructors, destructors and methods.

### Clause 5.3.4 Miscellaneous

The clause is to be extended:

`TriClassIdType` A value of type `TriClassIdType` is the name of a class as specified in the TTCN-3 ATS.

`TriObjHandleType` A value of type `TriObjHandle` contains platform-specific data allowing access to external objects.

## 6.2 Extensions to clause 5.6.3 of ETSI ES 201 873-5 Miscellaneous operations

### Clause 5.6.3.4 `triExternalCreate` (TE → PA)

This clause is to be added:

<b>Signature</b>	<code>TriStatusType triExternalCreate(     in TriClassIdType classId,     inout TriParameterListType parameterList,     out TriObjHandleType createdObject)</code>
<b>In Parameters</b>	<code>classId</code> identifier of the external class
<b>Out Parameters</b>	<code>returnValue</code> handle to the created object
<b>InOutParameters</b>	<code>parameterList</code> a list of encoded parameters for the indicated constructor. The parameters in <code>parameterList</code> are ordered as they appear in the TTCN-3 constructor declaration.
<b>Return Value</b>	The return status of the <code>triExternalCreate</code> operation. The return status indicates the local success ( <b>TRI_OK</b> ) or failure ( <b>TRI_Error</b> ) of the operation.
<b>Constraints</b>	This operation is called by the TE when it invokes a constructor specified in a class which is defined as external in TTCN-3. In the invocation of a <code>triExternalCreate</code> operation by the TE all <i>in</i> and <i>inout</i> constructor parameters contain encoded values. No error shall be indicated by the PA in case the value of any <i>out</i> parameter is non-null.
<b>Effect</b>	The PA shall implement the behaviour for each external class specified in the TTCN-3 ATS. On invocation of this operation, the PA shall invoke the constructor of a class indicated by the identifier <code>classId</code> . It shall access the specified <i>in</i> and <i>inout</i> constructor parameters in <code>parameterList</code> , create a new external object instance using the values of these parameters, and compute values for <i>inout</i> and <i>out</i> parameters in <code>parameterList</code> . The operation shall then return encoded values for all <i>inout</i> and <i>out</i> constructor parameters and a handle to the created external object. The <code>triExternalCreate</code> operation returns <b>TRI_OK</b> if the PA completes the constructor of the external class successfully, <b>TRI_Error</b> otherwise. In the latter case, the distinct value <code>null</code> shall be returned as the object handle. Note that whereas most of other TRI operations are considered to be non-blocking, the <code>triExternalCreate</code> operation is considered to be <i>blocking</i> . That means that the operation shall not return before the construction of the external object has been finished. External constructors have to be implemented carefully as they could cause deadlock of test component execution or even the entire test system implementation.

**Clause 5.6.3.5 triExternalFinally (TE → PA)**

This clause is to be added:

<b>Signature</b>	TriStatusType triExternalFinally( in TriObjHandleType handle)
<b>In Parameters</b>	handle handle to the object being destroyed
<b>Return Value</b>	The return status of the triExternalFinally operation. The return status indicates the local success ( <b>TRI_OK</b> ) or failure ( <b>TRI_Error</b> ) of the operation.
<b>Constraints</b>	This operation is called by the TE when it invokes a destructor specified in a class which is defined as external in TTCN-3.
<b>Effect</b>	The PA shall implement the behaviour for each external class specified in the TTCN-3 ATS which contains a destructor definition. On invocation of this operation, the PA shall invoke the destructor of the object whose handle is in the handle parameter. The triExternalFinally operation returns <b>TRI_OK</b> if the PA completes destruction of the external object successfully, <b>TRI_Error</b> otherwise. Note that whereas most of other TRI operations are considered to be non-blocking, the triExternalFinally operation is considered to be <i>blocking</i> . That means that the operation shall not return before the destruction of the external object has been finished. External destructors have to be implemented carefully as they could cause deadlock of test component execution or even the entire test system implementation.

**Clause 5.6.3.6 triExternalMethod (TE → PA)**

This clause is to be added:

<b>Signature</b>	TriStatusType triExternalMethod( in TriObjHandleType handle, in String methodName, inout TriParameterListType parameterList, out TriParameterType returnValue)
<b>In Parameters</b>	handle handle of the affected object; null for static methods methodName name of the called method
<b>Out Parameters</b>	returnValue (optional) encoded return value
<b>InOutParameters</b>	parameterList a list of encoded parameters for the indicated method. The parameters in parameterList are ordered as they appear in the TTCN-3 method declaration.
<b>Return Value</b>	The return status of the triExternalMethod operation. The return status indicates the local success ( <b>TRI_OK</b> ) or failure ( <b>TRI_Error</b> ) of the operation.
<b>Constraints</b>	This operation is called by the TE when it invokes a method specified in a class which is defined as external in TTCN-3. In the invocation of a triExternalMethod operation by the TE all <i>in</i> and <i>inout</i> parameters contain encoded values. No error shall be indicated by the PA in case the value of any <i>out</i> parameter is non-null.
<b>Effect</b>	The PA shall implement the behaviour for each method of all external classes specified in the TTCN-3 ATS. On invocation of this operation, the PA shall call a method methodName of an external object whose handle is in the handle parameter. It shall access the specified <i>in</i> and <i>inout</i> method parameters in parameterList, pass the values of these parameters to the called method, and compute values for <i>inout</i> and <i>out</i> parameters in parameterList. The operation shall then return encoded values for all <i>inout</i> and <i>out</i> method parameters and the encoded return value. If no return type has been defined for this method in the TTCN-3 ATS, the distinct value null shall be used for the latter. The triExternalMethod operation returns <b>TRI_OK</b> if the PA completes the method of the external object successfully, <b>TRI_Error</b> otherwise. Note that whereas most of other TRI operations are considered to be non-blocking, the triExternalMethod operation is considered to be <i>blocking</i> . That means that the operation shall not return before the method call has been finished. Methods of external classes have to be implemented carefully as they could cause deadlock of test component execution or even the entire test system implementation.





**Methods:**

- `setHandle`  
Sets the contained object.
- `getHandle`  
Gets the contained object.

**Clause 6.5.3.1 TriPlatformPA**

This clause is to be extended.

```
// TriPlatform
// TE -> PA
package org.etsi.ttcn.tri;
public interface TriPlatformPA {
    ...

    // Ref: TRI-Definition 5.6.3.4
    public TriStatus triExternalCreate(TriClassIdType classId,
        TriParameterList parameterList, TriObjHandleWrapper handle);

    // Ref: TRI-Definition 5.6.3.5
    public TriStatus triExternalFinally(Object handle);

    // Ref: TRI-Definition 5.6.3.6
    public TriStatus triExternalMethod(Object handle, String methodName,
        TriParameterList parameterList, TriParameter returnValue);
}

```

**Clause 6.5.3.2 TriPlatformTE**

This clause is to be extended.

```
// TriPlatform
// PA -> TE
package org.etsi.ttcn.tri;
public interface TriPlatformTE {
    ...

    // Ref: TRI-Definition 5.6.3.7
    public void triRaiseException(TriException exc);
}

```

## 6.4 Extensions to clause 7 of ETSI ES 201 873-5 ANSI C language mapping

**Clause 7.2.1 Abstract type mapping**

This clause is to be extended.

TRI ADT	ANSI C Representation	Notes and comments
...		
<code>TriClassIdType</code>	<code>QualifiedName</code>	
<code>TriObjectHandleType</code>	<code>typedef void * TriObjectHandle;</code>	

### Clause 7.2.4 TRI operation mapping

This clause is to be extended.

IDL Representation	ANSI C Representation
...	
TriStatusType triExternalCreate (in TriClassIdType classId, inout TriParameterListType parameterList, out TriObjHandleType createdObject)	TriStatus triExternalCreate (const TriClassId* classId, TriParameterList* parameterList, TriObjectHandle* handle)
TriStatusType triExternalFinally (in TriObjHandleType handle)	TriStatus triExternalFinally (TriObjectHandle handle)
TriStatusType triExternalMethod(in TriObjHandleType handle, in String methodName, inout TriParameterListType parameterList, out TriParameterType returnValue)	TriStatus triExternalFunction (TriClassId handle, char* methodName, TriParameterList* parameterList, TriParameter* returnValue)
void triRaiseException(in TriExceptionType exc)	void triRaiseException(const TriException* exc)

## 6.5 Extensions to clause 8 of ETSI ES 201 873-5 C++ language mapping

### Clause 8.5.19 TriClassId

This clause is to be added.

A value of type TriClassIdType represents the name of a class as specified in the TTCN-3 ATS. It is a derived class from QualifiedName, mapped to the following pure virtual class:

```
class TriClassId : public QualifiedName {
public:
    virtual ~TriClassId ();
    virtual Tboolean operator== (const TriClassId &sid) const =0;
    virtual TriClassId * cloneClassId () const =0;
    virtual Tboolean operator< (const TriClassId &sid) const =0;
}
```

#### Methods:

- ~TriClassId  
Destructor.
- operator==  
Returns true if both TriClassId objects are equal.
- cloneClassId  
Returns a copy of the TriClassId.
- operator<  
Operator < overload.

### Clause 8.5.20 TriObjectHandle

This clause is to be added.

A value of type TriObjectHandle type is mapped to a void pointer:

```
typedef void * TriObjectHandle;
```

**Clause 8.6.3 TriPlatformPA**

This clause is to be extended.

```
class TriPlatformPA {
public:
    ...

    //For each constructor on an external class specified in the TTCN-3 ATS implement the behaviour.
    virtual TriStatus triExternalCreate (const TriClassId *classId, TriParameterList
    *parameterList, TriObjectHandle * handle)=0;

    //For each destructor on an external class specified in the TTCN-3 ATS implement the behaviour.
    virtual TriStatus triExternalCreate (TriObjectHandle handle)=0;

    //For each method on an external class specified in the TTCN-3 ATS implement the behaviour.
    virtual TriStatus triExternalMethod (TriObjectHandle handle, const Tstring & methodName,
    TriParameterList *parameterList, TriParameter *returnValue)=0;
}
```

**Clause 8.6.4 TriPlatformTE**

This clause is to be extended.

```
class TriPlatformTE {
public:
    ...

    //Raises an exception during execution of external code in PA
    virtual void triRaiseException (const TriException *exc)=0;
}
```

## 6.6 Extensions to clause 9 of ETSI ES 201 873-5 C# language mapping

**Clause 9.4.2.19 TriClassId**

This clause is to be added.

**TriClassIdType** C# mapping is derived from the IQualifiedName interface:

```
public interface ITriClassId : IQualifiedName {}
```

**Clause 9.4.2.20 TriObjectHandleType mapping**

This clause is to be added.

**TriObjectHandleIdType** is mapped to the C# object class.

**Clause 9.5.2.3 TriPlatformPA**

This clause is to be extended.

```
public interface ITriPlatformPA {
    ...

    // Miscellaneous operations
    // Ref: TRI-Definition clause 5.6.3.4
    TriStatus TriExternalCreate(ITriClassId classId,
    ITriParameterList parameterList, out object handle);

    // Ref: TRI-Definition clause 5.6.3.5
    TriStatus TriExternalFinally(object handle);

    // Ref: TRI-Definition clause 5.6.3.6
    TriStatus TriExternalMethod(object handle, string methodName,
    ITriParameterList parameterList, ITriParameter returnValue);
}
```

### Clause 9.5.2.4 TriPlatformTE

This clause is to be extended.

```
public interface ITriPlatformTE {
    ...
    // Ref: TRI Definition clause 5.6.3.7
    void TriRaiseException(ITriException exc);
}
```

## 7 TCI Extensions for the Package

### 7.1 Extensions to clause 7.2.2.1 of ETSI ES 201 873-6 Abstract TTCN-3 data types and values

The definition of the getTypeClass operation shall be modified of the following way:

TciTypeClassType getTypeClass()	Returns the type class of the respective type. A value of TciTypeClassType can have one of the following constants: ADDRESS, ANYTYPE, ARRAY, BITSTRING, BOOLEAN, CHARSTRING, COMPONENT, ENUMERATED, FLOAT, HEXSTRING, INTEGER, OCTETSTRING, RECORD, RECORD_OF, SET, SET_OF, UNION, UNIVERSAL_CHARSTRING, VERDICT, DEFAULT, PORT, TIMER, CLASS.
---------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### 7.2 Extensions to clause 7.2.2 of ETSI ES 201 873-6 Abstract TTCN-3 data types and values

#### Clause 7.2.2.5 Abstract TTCN-3 classes

This clause is to be added.

According to the present document, TTCN-3 user-defined classes will be represented at the TCI interfaces using the abstract data type Class. The abstract data type Class is based on the abstract data type Type.

The following operations defined for abstract data type Type work differently in the abstract data type Class:

Value newInstance()	The method creates an ObjectInstance containing a reference to a null object of the class.
---------------------	--------------------------------------------------------------------------------------------

The following operations are defined for abstract data type Class:

ObjectInstance create(TriComponentIdType c, TciParameterListType tciPars)	Calls the constructor to create a new instance of this class using the supplied parameters for the specified component. Created objects are always considered to be initialized.
ClassSeq getSuperclasses ()	Returns the superclasses of the current class.
TStringSeq getFieldNames ()	Returns the names of all public fields defined in the class.
TStringSeq getMethodNames ()	Returns the names of all public methods of the class.

`TciParameterTypeListType` `getConstructorParameters ()`  
Returns formal parameters of the class constructor.

`TciParameterTypeListType` `getMethodParameters (TString methodName)`  
Returns formal parameters of the specified public method. The distinct value `null` is returned if the method does not exist or is not public.

`Type` `getFieldType (TString name)`  
Returns the type of the specified public field. The distinct value `null` is returned if the member variable does not exist or is not public.

`Type` `getMethodReturnType (TString name)`  
Returns the return type of specified public method or the distinct value `null` if no return value is defined, the method does not exist or it is not public.

#### **Clause 7.2.2.6 ClassSeq**

This clause is to be added.

The abstract data type `ClassSeq` is used to represent a list of classes.

## 7.3 Extensions to clause 7.2.2.2.0 of ETSI ES 201 873-6 Basic rules

The figure 4 is to be extended:

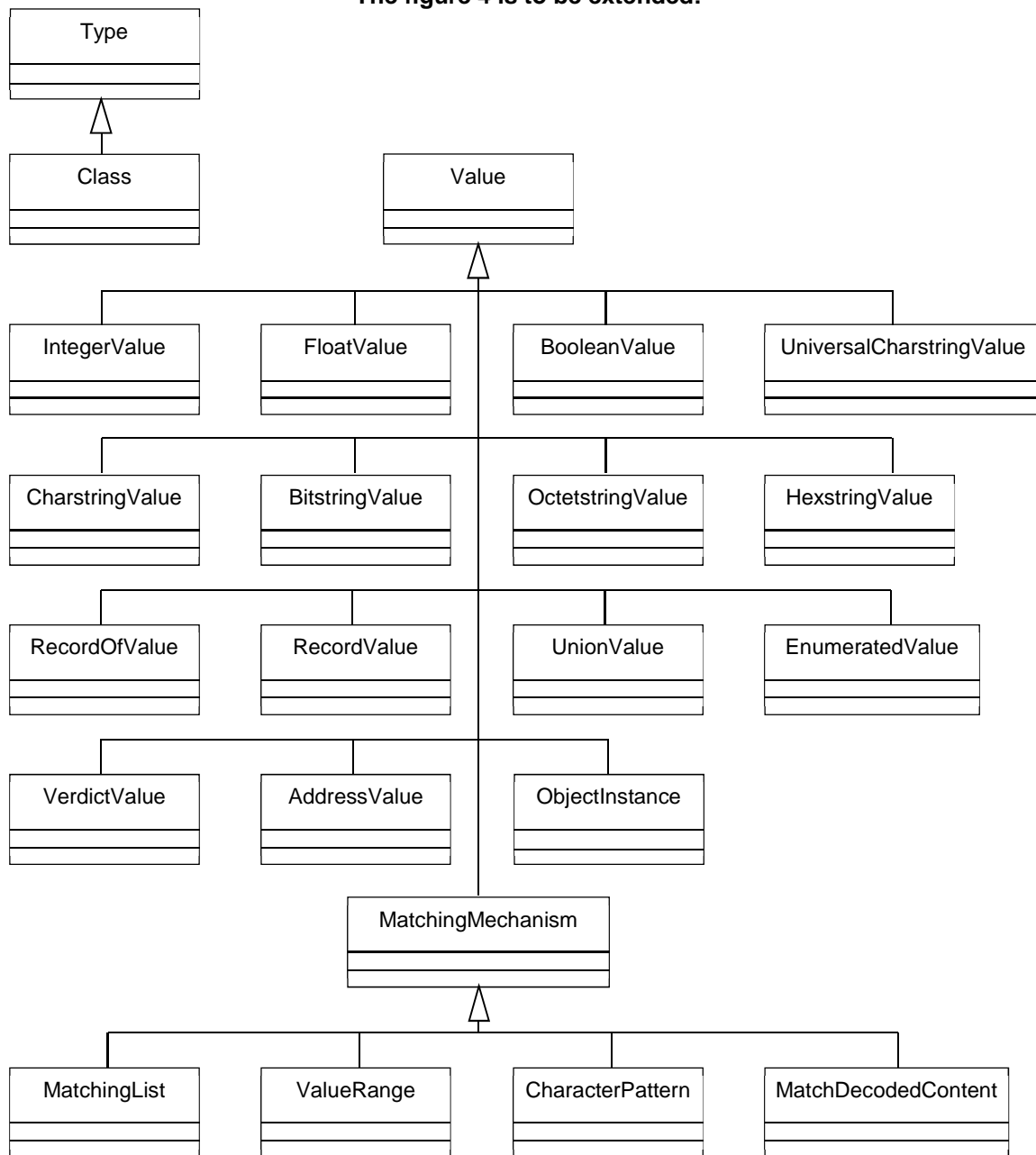


Figure 1: Hierarchy of abstract values

## 7.4 Extensions to clause 7.2.2.2 of ETSI ES 201 873-6 Abstract TTCN-3 values

### Clause 7.2.2.16 The abstract data type ObjectInstance

This clause is to be added.

The abstract data type `ObjectInstance` is based on the abstract data type `Value`. It is used to modify the referenced object and to access public object fields and methods.

The following operations are defined on the abstract data type `ObjectInstance`:

`TriComponentIdType getOwner ()` Returns the component that owns the object instance.

`TString getId ()` Returns an identifier of the object which is unique within the owner component context.

`void setObject (ObjectInstance source)`  
The operation sets the referenced object to be the same one as the one referenced by the `source` parameter. In case the source object does not contain a null reference, the object instance and the source object shall be owned by the same component.

`Value getField (TString fieldName)`  
Returns the value of the referenced public member field. The distinct value `null` is returned if the object does not contain the referenced field or the field is not accessible.

`Value callMethod(TString methodName, TciParameterListType tciPars)`  
Calls the method of the object instance. The distinct value `null` is returned if the method does not return any value. A runtime error is generated if the method does not exist or if the given parameters do not conform to the formal parameters of the declared method.

## 7.5 Extensions to clause 7.3.4.1 of ETSI ES 201 873-6 Abstract TCI-TL provided

### Clause 7.3.4.1.122 `tliObjCreateEnter`

This clause is to be added.

<b>Signature</b>	<code>void tliObjCreateEnter(in TString am, in TInteger ts, in TString src, in TInteger line, in TriComponentIdType c, in ObjectInstance obj, in TciParameterListType tciPars)</code>	
<b>In Parameters</b>	<code>am</code>	An additional message.
	<code>ts</code>	The time when the event is produced.
	<code>src</code>	The source file of the test specification.
	<code>line</code>	The line number where the request is performed.
	<code>c</code>	The component which produces this event.
	<code>obj</code>	The object being created.
	<code>tciPars</code>	The parameters of the constructor.
<b>Return Value</b>	<code>Void</code>	
<b>Constraint</b>	Shall be called by TE to log the entering of a constructor of an object. This event occurs after the constructor has been entered.	
<b>Effect</b>	The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document.	

### Clause 7.3.4.1.123 `tliObjCreateLeave`

This clause is to be added.

<b>Signature</b>	<code>void tliObjCreateLeave(in TString am, in TInteger ts, in TString src, in TInteger line, in TriComponentIdType c, in ObjectInstance obj, in TciParameterListType tciPars)</code>	
<b>In Parameters</b>	<code>am</code>	An additional message.
	<code>ts</code>	The time when the event is produced.
	<code>src</code>	The source file of the test specification.
	<code>line</code>	The line number where the request is performed.
	<code>c</code>	The component which produces this event.
	<code>obj</code>	The created object instance.
	<code>tciPars</code>	The parameters of the constructor.
<b>Return Value</b>	<code>Void</code>	
<b>Constraint</b>	Shall be called by TE to log the leaving of an object constructor. This event occurs after the constructor has been left.	

<b>Effect</b>	The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document.
---------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------

#### Clause 7.3.4.1.124 tliObjFinallyEnter

This clause is to be added.

<b>Signature</b>	void tliObjFinallyEnter(in TString am, in TInteger ts, in TString src, in TInteger line, in TriComponentIdType c, in ObjectInstance obj)	
<b>In Parameters</b>	am	An additional message.
	ts	The time when the event is produced.
	src	The source file of the test specification.
	line	The line number where the request is performed.
	c	The component which produces this event.
	obj	The object instance being destroyed.
<b>Return Value</b>	Void	
<b>Constraint</b>	Shall be called by TE to log the entering of a destructor of an object. This event occurs after the destructor has been entered.	
<b>Effect</b>	The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document.	

#### Clause 7.3.4.1.125 tliObjFinallyLeave

This clause is to be added.

<b>Signature</b>	void tliObjCreateLeave(in TString am, in TInteger ts, in TString src, in TInteger line, in TriComponentIdType c, in ObjectInstance obj, in TciParameterListType tciPars)	
<b>In Parameters</b>	am	An additional message.
	ts	The time when the event is produced.
	src	The source file of the test specification.
	line	The line number where the request is performed.
	c	The component which produces this event.
	obj	The object being destroyed.
<b>Return Value</b>	Void	
<b>Constraint</b>	Shall be called by TE to log the leaving of an object destructor. This event occurs after the destructor has been left. Accessing any members, properties and methods of a destroyed object with exception of methods used for comparison shall cause an error.	
<b>Effect</b>	The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document.	

#### Clause 7.3.4.1.126 tliObjMethodEnter

This clause is to be added.

<b>Signature</b>	void tliObjMethodEnter(in TString am, in TInteger ts, in TString src, in TInteger line, in TriComponentIdType c, in ObjectInstance obj, in TString methodName, in TciParameterListType tciPars)	
<b>In Parameters</b>	Am	An additional message.
	Ts	The time when the event is produced.
	src	The source file of the test specification.
	line	The line number where the request is performed.
	C	The component which produces this event.
	obj	The affected object instance.
	methodName	The name of the called method.
	tciPars	The parameters of the called method.
<b>Return Value</b>	void	
<b>Constraint</b>	Shall be called by TE to log the entering of an object method. This event occurs after the method has been entered.	
<b>Effect</b>	The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document.	



**Clause 7.3.4.1.127 tliObjMethodLeave**

This clause is to be added.

<b>Signature</b>	void tliObjMethodLeave(in TString am, in TInteger ts, in TString src, in TInteger line, in TriComponentIdType c, in ObjectInstance obj, in TString methodName, in TciParameterListType tciPars, in Value returnValue)	
<b>In Parameters</b>	Am	An additional message.
	Ts	The time when the event is produced.
	src	The source file of the test specification.
	line	The line number where the request is performed.
	C	The component which produces this event.
	obj	The affected object instance.
	methodName	The name of the called method.
	tciPars	The parameters of the called method.
<b>Return Value</b>	void	
<b>Constraint</b>	Shall be called by TE to log the leaving of an object method. This event occurs after the method has been left.	
<b>Effect</b>	The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document.	

**Clause 7.3.4.1.132 tliObjVar**

This clause is to be added.

<b>Signature</b>	void tliObjVar(in TString am, in TInteger ts, in TString src, in TInteger line, in TriComponentIdType c, in ObjectInstance obj, in TString name, in Value value)	
<b>In Parameters</b>	Am	An additional message.
	Ts	The time when the event is produced.
	Src	The source file of the test specification.
	Line	The line number where the request is performed.
	C	The component which produces this event.
	Obj	The affected object instance.
	name	The name of the member variable.
	value	The new value of the member variable.
<b>Return Value</b>	Void	
<b>Constraint</b>	Shall be called by TE to log the modification of the value of a field of an object. This event occurs after the field value has been changed. In case of @lazy fields, it is called also after performing evaluation as the evaluation result is automatically assigned to the field.	
<b>Effect</b>	The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document.	

## 7.6 Extensions to clause 8 of ETSI ES 201 873-6 Java™ language mapping

**Clause 8.3.2.4 TciTypeClassType**

This clause is to be extended.

**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 TString      getId ();
public void         setObject (ObjectInstance source);
public Value        callMethod (String methodName, TciParameterList tciPars);
}

```

**Methods:**

- `getOwner` Returns the component that owns the object instance.
- `getId` Returns the unique identifier of the object instance.
- `setObject` The operation sets the referenced object to the same reference as the given object.
- `getField` Gets the value of the referenced public field.
- `callMethod` Calls a method of the object instance.

**Clause 8.5.4.1 TCI-TL provided**

The TciTLProvided interface is to be extended:

```

package org.etsi.ttcn.tci;
public interface TciTLProvided {
    ...
    public void tliObjCreateEnter(String am, int ts, String src, int line, TriComponentId c,
        ObjectInstance obj, TciParameterList tciPars);
    public void tliObjCreateLeave(String am, int ts, String src, int line, TriComponentId c,
        ObjectInstance obj, TciParameterList tciPars);
    public void tliObjFinallyEnter(String am, int ts, String src, int line, TriComponentId c,
        ObjectInstance obj);
    public void tliObjFinallyLeave(String am, int ts, String src, int line, TriComponentId c,
        ObjectInstance obj);
    public void tliObjMethodEnter(String am, int ts, String src, int line, TriComponentId c,
        ObjectInstance obj, String methodName, TciParameterList tciPars);
    public void tliObjMethodLeave(String am, int ts, String src, int line, TriComponentId c,
        ObjectInstance obj, String methodName, TciParameterList tciPars, Value returnValue);
    public void tliObjVar(String am, int ts, String src, int line, TriComponentId c,
        ObjectInstance obj, String name, Value value);
}

```

## 7.7 Extensions to clause 9 of ETSI ES 201 873-6 ANSI C language mapping

**Clause 9.2 Data**

The table 5 is to be extended.

TCI IDL Interface	ANSI C representation	Notes and comments
:		
<b>Class</b>		
Value create(TriComponentIdType c, TciParameterListType tciPars)	Value tciObjCreate(Type cls, TriComponentId c, TciParameterListType tciPars)	
ClassSeq getSuperclasses ()	Type* tciGetSuperclasses (Type cls)	Returns null pointer or a null-pointer terminated array
TStringSeq getFieldNames ()	String* tciGetClassFieldNames (Type cls)	Returns null pointer or a null-pointer terminated array
TStringSeq getMethodNames ()	String* tciGetClassMethodNames (Type cls)	Returns null pointer or a null-pointer terminated array
TciParameterTypeListType getConstructorParameters ()	TciParameterTypeListType* tciGetClassConstructorParameters (Type cls)	
TciParameterTypeListType getMethodParameters (TString methodName)	TciParameterTypeListType* tciGetClassMethodParameters (Type cls, String methodName)	
Type getMemberType (TString name)	Type tciGetClassFieldType(Type cls, String name)	

TCI IDL Interface	ANSI C representation	Notes and comments
Type getMethodReturnType (TString methodName)	Type tciGetClassMethodReturnType (Type cls, String methodName)	
<b>ObjectInstance</b>		
TriComponentIdType getOwner ()	TriComponentId tciGetObjOwner (Value obj)	
TString getId ()	char * tciGetObjUniqueId (Value obj)	
void setObject (ObjectInstance source)	void tciSetObject (Value obj, Value source)	
Value getField (TString fieldName)	Value tciGetObjField (Value obj, String fieldName)	
Value callMethod(TString methodName, TciParameterListType tciPars)	Value tciCallObjMethod(Value obj, String methodName, TciParameterListType tciPars)	

#### Clause 9.4.4.1 TCI-TL provided

The clause is to be extended.

```

void tliObjCreateEnter
(String am, int ts, String src, int line, TriComponentId c, Value obj,
 TciParameterListType tciPars);
void tliObjCreateLeave
(String am, int ts, String src, int line, TriComponentId c, Value obj,
 TciParameterListType tciPars);
void tliObjFinallyEnter
(String am, int ts, String src, int line, TriComponentId c, Value obj);
void tliObjFinallyLeave
(String am, int ts, String src, int line, TriComponentId c, Value obj);
void tliObjMethodEnter
(String am, int ts, String src, int line, TriComponentId c, Value obj, String methodName,
 TciParameterListType tciPars);
void tliObjMethodLeave
(String am, int ts, String src, int line, TriComponentId c, Value obj, String methodName,
 TciParameterListType tciPars, Value returnValue);
void tliObjVar
(String am, int ts, String src, int line, TriComponentId c, Value obj, String name,
 Value value);

```

#### Clause 9.5 Data

The definition of the TciTypeClassType in the table 7 is to be modified.

TCI IDL ADT	ANSI C representation (Type definition)	Notes and comments
:		
TciTypeClassType	<pre> 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_CLASS_TYPE = 25 } TciTypeClassType; </pre>	
:		

## 7.8 Extensions to clause 10 of ETSI ES 201 873-6 C++ language mapping

### Clause 10.5.2.14 TciTypeClass

This clause is to be extended.

```
typedef enum
{
    TCI_ADDRESS = 0,
    TCI_ANYTYPE = 1,
    TCI_BITSTRING = 2,
    TCI_BOOLEAN = 3,
    TCI_CHARSTRING = 5,
    TCI_COMPONENT = 6,
    TCI_ENUMERATED = 7,
    TCI_FLOAT = 8,
    TCI_HEXSTRING = 9,
    TCI_INTEGER = 10,
    TCI_OCTETSTRING = 12,
    TCI_RECORD = 13,
    TCI_RECORD_OF = 14,
    TCI_ARRAY = 15,
    TCI_SET = 16,
    TCI_SET_OF = 17,
    TCI_UNION = 18,
    TCI_UNIVERSAL_CHARSTRING = 20,
    TCI_VERDICT = 21,
    TCI_DEFAULT = 22,
    TCI_PORT = 23,
    TCI_TIMER = 24,
    TCI_CLASS = 25
} TciTypeClass;
```

### Clause 10.5.3.23 Class

This clause is to be added.

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

```
class TciClass : public virtual TciType {
public:
    virtual ~TciClass ();
    virtual ObjectInstance * create(const TriComponentId & c, TciParameterList & tciPars) =0;
    virtual const std::vector<TciClass*> & getSuperclasses () const =0;
    virtual const std::vector<Tstring*> & getFieldNames () const =0;
    virtual const std::vector<Tstring*> & getMethodNames () const =0;
    virtual const TciParameterTypeList & getConstructorParameters () const =0;
    virtual const TciParameterTypeList & getMethodParameters (Tstring methodName) const =0;
    virtual const TciType & getMemberType (const Tstring & name) const =0;
    virtual const TciType & getMethodReturnValue (const Tstring & name) const =0;
    virtual Tboolean operator== (const TciClass &p_class) const =0;
    virtual TciClass * clone () const =0;
    virtual Tboolean operator< (const TciClass &p_content) const =0;
}
```

#### Methods:

~TciClass

Destructor

create

Calls the constructor to create a new instance of this class using the supplied parameters for the specified component

getSuperclasses

Returns the superclasses of the current class

getFieldNames

Returns the names of all public fields defined in the class

getMethodNames

Returns the names of all public methods of the class

getConstructorParameters

Returns formal parameters of the class constructor

getMethodParameters  
Returns formal parameters of the specified public method

getFieldType  
Returns the type of the specified public field

getMethodReturnValue  
Returns the return type of specified public method or the distinct value null if no return value is defined

operator==  
Returns true if both objects are equal

clone  
Return a copy of the matching mechanism

operator<  
Operator < overload

### Clause 10.5.3.24 ObjectInstance

This clause is to be added.

TTCN-3 implication and exclusion matching mechanism support. It is mapped to the following pure virtual class:

```
class ObjectInstance : public virtual TciValue {
public:
    virtual ~ObjectInstance ();
    virtual const TriComponentId & getOwner () const =0;
    virtual const TString getId () const =0;
    virtual void setObject (ObjectInstance & val) =0;
    virtual TciValue * getField (const TString & fieldName) =0;
    virtual Value callMethod(const TString & methodName, TciParameterList & tciPars) =0;
    virtual Tboolean operator== (const ObjectInstance &p_obj) const =0;
    virtual ObjectInstance * clone () const =0;
    virtual Tboolean operator< (const ObjectInstance &p_content) const =0;
}
```

#### Methods:

~ObjectInstance  
Destructor

getOwner  
Returns the component that owns the object instance

getId  
Returns the unique identifier of the object instance

setObject  
The operation sets the referenced object

getField  
Returns the value of the referenced public field

callMethod  
Calls a method of the object instance

operator==  
Returns true if both objects are equal

clone  
Return a copy of the matching mechanism

operator<  
Operator < overload

### Clause 10.5.3.25 ClassSeq

This clause is to be added.

The ClassSeq abstract data type is mapped to `std::vector<TciClass*>`.

### Clause 10.6.4.1 TciTIPProvided

This clause is to be extended.

```
//Called by TE to log the entering of a constructor
virtual void tliObjCreateEnter (const TString &am, const timeval ts, const TString &src, const
Tinteger line, const TriComponentId *c, const ObjectInstance *obj, const TciParameterList
*tciPars)=0;

//Called by TE to log the leaving of a constructor
virtual void tliObjCreateLeave (const TString &am, const timeval ts, const TString &src, const
Tinteger line, const TriComponentId *c, const ObjectInstance *obj, const TciParameterList
*tciPars)=0;
```

```

//Called by TE to log the entering of a destructor
virtual void tliObjFinallyEnter (const Tstring &am, const timeval ts, const Tstring &src, const
Tinteger line, const TriComponentId *c, const ObjectInstance *obj)=0;

//Called by TE to log the leaving of a destructor
virtual void tliObjFinallyLeave (const Tstring &am, const timeval ts, const Tstring &src, const
Tinteger line, const TriComponentId *c, const ObjectInstance *obj)=0;

//Called by TE to log the entering of an object method
virtual void tliObjMethodEnter (const Tstring &am, const timeval ts, const Tstring &src, const
Tinteger line, const TriComponentId *c, const ObjectInstance *obj, const Tstring &methodName, const
TciParameterList *tciPars)=0;

//Called by TE to log the leaving of an object method
virtual void tliObjMethodLeave (const Tstring &am, const timeval ts, const Tstring &src, const
Tinteger line, const TriComponentId *c, const ObjectInstance *obj, const Tstring &methodName, const
TciParameterList *tciPars, const TciValue *returnValue)=0;

//Called by TE to log the modification of a member variable of an object
virtual void tliObjVar (const Tstring &am, const timeval ts, const Tstring &src, const Tinteger
line, const TriComponentId *c, const ObjectInstance *obj, const Tstring &name, const TciValue
*value)=0;

```

## 7.9 Extensions to clause 11 of ETSI ES 201 873-6 W3C XML mapping

### Clause 11.3.3.30 ObjectInstance

ObjectInstance type is mapped to the complex type specified below. The content of the XML elements based on the ObjectInstance type shall be equal to the string produced by the valueToString operation (described in clause 7.2.2.2.1 of ETSI ES 201 873-6):

```

<xsd:complexType name="ObjectInstance">
  <xsd:group ref="Values:BaseValue"/>
  <xsd:attributeGroup ref="Values:ValueAtts"/>
</xsd:complexType>

```

#### Items:

- BaseValue                      Object instance content described in clause 11.3.3.1 of ETSI ES 2001-873-6
- ValueAtts                      Value attributes described in clause 11.3.3.1 of ETSI ES 2001-873-6

### Clause 11.4.2.1 TCI-TL provided

This clause is to be extended.

```

<xsd:complexType name="tliObjCreateEnter">
  <xsd:complexContent mixed="true">
    <xsd:extension base="Events:Event">
      <xsd:sequence>
        <xsd:element name="obj" type="Values:ObjectInstance" />
        <xsd:element name="tciPars" type="Types:TciParameterListType" minOccurs="0"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="tliObjCreateLeave">
  <xsd:complexContent mixed="true">
    <xsd:extension base="Events:Event">
      <xsd:sequence>
        <xsd:element name="obj" type="Values:ObjectInstance" />
        <xsd:element name="tciPars" type="Types:TciParameterListType" minOccurs="0"/>
        <xsd:element name="returnValue" type="Values:Value" minOccurs="0"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="tliObjFinallyEnter">
  <xsd:complexContent mixed="true">

```

```

    <xsd:extension base="Events:Event">
      <xsd:sequence>
        <xsd:element name="obj" type="Values:ObjectInstance" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="tliObjFinallyLeave">
  <xsd:complexContent mixed="true">
    <xsd:extension base="Events:Event">
      <xsd:sequence>
        <xsd:element name="obj" type="Values:ObjectInstance" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="tliObjMethodEnter">
  <xsd:complexContent mixed="true">
    <xsd:extension base="Events:Event">
      <xsd:sequence>
        <xsd:element name="obj" type="Values:ObjectInstance" />
        <xsd:element name="methodName" type="SimpleTypes:TString" />
        <xsd:element name="tciPars" type="Types:TciParameterListType" minOccurs="0"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="tliObjMethodLeave">
  <xsd:complexContent mixed="true">
    <xsd:extension base="Events:Event">
      <xsd:sequence>
        <xsd:element name="obj" type="Values:ObjectInstance" />
        <xsd:element name="methodName" type="SimpleTypes:TString" />
        <xsd:element name="tciPars" type="Types:TciParameterListType" minOccurs="0"/>
        <xsd:element name="returnValue" type="Values:Value" minOccurs="0"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="tliObjVar">
  <xsd:complexContent mixed="true">
    <xsd:extension base="Events:Event">
      <xsd:sequence>
        <xsd:element name="obj" type="Values:ObjectInstance" />
        <xsd:element name="name" type="SimpleTypes:TString" />
        <xsd:element name="val" type="Values:Value" minOccurs="0"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

```

## 7.10 Extensions to clause 12 of ETSI ES 201 873-6 C# language mapping

### Clause 12.4.2.4 TciTypeClassType

This clause is to be extended.

**TciTypeClassType** is mapped to the following enumeration:

```

public enum TciTypeClass {
  Address = 0,
  Anytype = 1,
  Bitstring = 2,
  BooleanType = 3,
  Charstring = 5,
  Component = 6,
  Enumerated = 7,
  Float = 8,
  Hexstring = 9,

```





```

    ITciValue    CallMethod (String methodName, ITciParameterList tciPars);
}

```

#### Methods:

- Owner Returns the component that owns the object instance.
- Id Returns the unique identifier of the object instance.
- SetObject The operation sets the referenced object.
- GetField Returns the value of the referenced public field.
- CallMethod Calls a method of the object instance.

#### Clause 12.5.4.1 TCI-TL provided

The ITciTLProvided interface is to be extended:

```

public interface ITciTLProvided {
    ...
    void TliObjCreateEnter(string am, System.DateTime ts, string src, int line,
        ITriComponentId c, ITciObjectInstance obj, ITciParameterList tciPars);
    void TliObjCreateLeave(string am, System.DateTime ts, string src, int line,
        ITriComponentId c, ITciObjectInstance obj, ITciParameterList tciPars);
    void TliObjFinallyEnter(string am, System.DateTime ts, string src, int line,
        ITriComponentId c, ITciObjectInstance obj);
    void TliObjFinallyLeave(string am, System.DateTime ts, string src, int line,
        ITriComponentId c, ITciObjectInstance obj);
    void TliObjMethodEnter(string am, System.DateTime ts, string src, int line,
        ITriComponentId c, ITciObjectInstance obj, string methodName, ITciParameterList tciPars);
    void TliObjMethodLeave(string am, System.DateTime ts, string src, int line,
        ITriComponentId c, ITciObjectInstance obj, string methodName, ITciParameterList tciPars,
        ITciValue returnValue);
    void TliObjVar (string am, System.DateTime ts, string src, int line,
        ITriComponentId c, ITciObjectInstance obj, string name, ITciValue value);
}

```

## 8 XTRI Extensions for the Package (optional)

### 8.1 Changes to clause 5.6.3 of ETSI ES 201 873-5 Miscellaneous operations

#### Clause 5.6.3.4 triExternalCreate → xtriExternalCreate

<b>Signature</b>	TriStatusType xtriExternalCreate( in TriClassIdType classId, inout TciParameterListType parameterList, out TriObjHandleType createdObject)
<b>In Parameters</b>	classId            identifier of the external class
<b>Out Parameters</b>	createdObject    handle to the created object
<b>InOutParameters</b>	parameterList    a list of <del>encoded</del> parameters for the indicated constructor. The parameters in parameterList are ordered as they appear in the TTCN-3 constructor declaration.
<b>Return Value</b>	The return status of the xtriExternalCreate operation. The return status indicates the local success ( <b>TRI_OK</b> ) or failure ( <b>TRI_Error</b> ) of the operation.
<b>Constraints</b>	This operation is called by the TE when it invokes a constructor specified in a class which is defined as external in TTCN-3. In the invocation of a triExternalCreate operation by the TE all <i>in</i> and <i>inout</i> constructor parameters contain <del>encoded</del> values. No error shall be indicated by the PA in case the value of any <i>out</i> parameter is non-null.

<b>Effect</b>	<p>The PA shall implement the behaviour for each external class specified in the TTCN-3 ATS. On invocation of this operation, the PA shall invoke the constructor of a class indicated by the identifier <code>classId</code>. It shall access the specified <i>in</i> and <i>inout</i> constructor parameters in <code>parameterList</code>, create a new external object instance using the values of these parameters, and compute values for <i>inout</i> and <i>out</i> parameters in <code>parameterList</code>. The operation shall then return <del>encoded</del> values for all <i>inout</i> and <i>out</i> constructor parameters and a handle to the created external object.</p> <p>The <code>xtriExternalCreate</code> operation returns <b>TRI_OK</b> if the PA completes the constructor of the external class successfully, <b>TRI_Error</b> otherwise. In the latter case, the distinct value <code>null</code> shall be returned as the object handle.</p> <p>Note that whereas most of other TRI operations are considered to be non-blocking, the <code>xtriExternalCreate</code> operation is considered to be <i>blocking</i>. That means that the operation shall not return before the construction of the external object has been finished. External constructors have to be implemented carefully as they could cause deadlock of test component execution or even the entire test system implementation.</p>
---------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

#### Clause 5.6.3.6 triExternalMethod → xtriExternalMethod

<b>Signature</b>	<pre>TriStatusType xtriExternalMethod(     in TriObjHandleType handle,     in String methodName,     inout TciParameterListType parameterList,     out TciParameterType returnValue)</pre>				
<b>In Parameters</b>	<table> <tr> <td><code>handle</code></td> <td>handle of the affected object; null for static methods</td> </tr> <tr> <td><code>methodName</code></td> <td>name of the called method</td> </tr> </table>	<code>handle</code>	handle of the affected object; null for static methods	<code>methodName</code>	name of the called method
<code>handle</code>	handle of the affected object; null for static methods				
<code>methodName</code>	name of the called method				
<b>Out Parameters</b>	<code>returnValue</code> (optional) <del>encoded</del> return value				
<b>InOutParameters</b>	<code>parameterList</code> a list of <del>encoded</del> parameters for the indicated method. The parameters in <code>parameterList</code> are ordered as they appear in the TTCN-3 method declaration.				
<b>Return Value</b>	The return status of the <code>xtriExternalMethod</code> operation. The return status indicates the local success ( <b>TRI_OK</b> ) or failure ( <b>TRI_Error</b> ) of the operation.				
<b>Constraints</b>	<p>This operation is called by the TE when it invokes a method specified in a class which is defined as external in TTCN-3.</p> <p>In the invocation of a <code>xtriExternalMethod</code> operation by the TE all <i>in</i> and <i>inout</i> parameters contain <del>encoded</del> values. No error shall be indicated by the PA in case the value of any <i>out</i> parameter is non-null.</p>				
<b>Effect</b>	<p>The PA shall implement the behaviour for each method of all external classes specified in the TTCN-3 ATS. On invocation of this operation, the PA shall call a method <code>methodName</code> of an external object whose handle is in the <code>handle</code> parameter. It shall access the specified <i>in</i> and <i>inout</i> method parameters in <code>parameterList</code>, pass the values of these parameters to the called method, and compute values for <i>inout</i> and <i>out</i> parameters in <code>parameterList</code>. The operation shall then return <del>encoded</del> values for all <i>inout</i> and <i>out</i> method parameters and the <del>encoded</del> return value. If no return type has been defined for this method in the TTCN-3 ATS, the distinct value <code>null</code> shall be used for the latter.</p> <p>The <code>xtriExternalMethod</code> operation returns <b>TRI_OK</b> if the PA completes the method of the external object successfully, <b>TRI_Error</b> otherwise.</p> <p>Note that whereas most of other TRI operations are considered to be non-blocking, the <code>xtriExternalMethod</code> operation is considered to be <i>blocking</i>. That means that the operation shall not return before the method call has been finished. Methods of external classes have to be implemented carefully as they could cause deadlock of test component execution or even the entire test system implementation.</p>				

#### Clause 5.6.3.7 triRaiseException → xtriRaiseException

<b>Signature</b>	<code>void xtriRaiseException(in Value exc)</code>
<b>In Parameters</b>	<code>exc</code> <del>encoded</del> exception to raise
<b>Constraints</b>	This operation can be called by the PA during execution of <code>triExternalFunction</code> , <code>triExternalCreate</code> , <code>triExternalFinally</code> or <code>triExternalMethod</code> .
<b>Effect</b>	The operation raises an exception that can be later processed by the TE in the <code>catch</code> statement. <del>The exception is provided in an encoded form. Decoding is performed in the catch statement.</del>

## 8.2 Extensions to clause 6 of ETSI ES 201 873-5 Java™ language mapping

### Clause 6.5.3.1 Changes to TriPlatformPA

This clause is to be extended.

```
// TriPlatform
// TE -> PA
package org.etsi.ttcn.tri;
public interface xTriPlatformPA {
    ...

    // Ref: TRI-Definition 5.6.3.4
    public TriStatus triExternalCreate(TriClassIdType classId,
        TciParameterList parameterList, TriObjHandleWrapper handle);

    // Ref: TRI-Definition 5.6.3.6
    public TriStatus xtriExternalMethod(Object handle, String methodName,
        TciParameterList parameterList, TciParameter returnValue);
}
```

### Clause 6.5.3.1 Changes to TriPlatformPA

This clause is to be extended.

```
// TriPlatform
// PA -> TE
package org.etsi.ttcn.tri;
public interface xTriPlatformTE {
    ...

    // Ref: TRI-Definition 5.6.3.7
    public void triRaiseException(Value exc);
}
```

## 8.3 Extensions to clause 7 of ETSI ES 201 873-5 ANSI C language mapping

### Clause 7.2.4 TRI operation mapping

This clause is to be extended.

IDL Representation	ANSI C Representation
...	
TriStatusType xtriExternalCreate (in TriClassIdType classId, inout TciParameterListType parameterList, out TriObjHandleType createdObject)	TriStatus xtriExternalCreate (const TriClassId* classId, TciParameterList* parameterList, TriObjectHandle* handle)
TriStatusType xtriExternalMethod(in TriObjHandleType handle, in String methodName, inout TciParameterListType parameterList, out TciParameterType returnValue)	TriStatus xtriExternalFunction (TriClassId handle, char* methodName, TciParameterList* parameterList, TciParameter* returnValue)
void xtriRaiseException(in Value exc)	void xtriRaiseException(const Value* exc)

## 8.4 Extensions to clause 8 of ETSI ES 201 873-5 C++ language mapping

### Clause 8.6.3 TriPlatformPA

This clause is to be extended.

```
class xTriPlatformPA {
public:
    ...

    //For each constructor on an external class specified in the TTCN-3 ATS implement the behaviour.
    virtual TriStatus xtriExternalCreate (const TriClassId *classId, TciParameterList
    *parameterList, TriObjectHandle * handle)=0;
    //For each method on an external class specified in the TTCN-3 ATS implement the behaviour.
    virtual TriStatus xtriExternalMethod (TriObjectHandle handle, const Tstring & methodName,
    TciParameterList *parameterList, TciParameter *returnValue)=0;
}

```

### Clause 8.6.4 TriPlatformTE

This clause is to be extended.

```
class xTriPlatformTE {
public:
    ...

    //Raises an exception during execution of external code in PA
    virtual void xtriRaiseException (const TciValue *exc)=0;
}

```

## 8.5 Extensions to clause 9 of ETSI ES 201 873-5 C# language mapping

### Clause 9.5.2.3 TriPlatformPA

This clause is to be extended.

```
public interface IXTriPlatformPA {
    ...

    // Miscellaneous operations
    // Ref: TRI-Definition clause 5.6.3.4
    TriStatus XTriExternalCreate(ITriClassId classId,
        ITciParameterList parameterList, out object handle);

    // Ref: TRI-Definition clause 5.6.3.6
    TriStatus XTriExternalMethod(object handle, string methodName,
        ITciParameterList parameterList, ITciParameter returnValue);
}

```

### Clause 9.5.2.4 TriPlatformTE

This clause is to be extended.

```
public interface IXTriPlatformTE {
    ...

    // Ref: TRI Definition clause 5.6.3.7
    void XTriRaiseException(ITciValue exc);
}

```

## Annex A (normative): BNF and static semantics

### A.1 Extensions to TTCN-3 terminals

The list of reserved terminals which are keywords in table A.3 in ETSI ES 201 873-1 [1] needs to be extended with **class**, **finally**, **object**, and **this**. The extension of table A.3 in ETSI ES 201 873-1 [1], clause A.1.5.0 is shown below.

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

## A.2 Modified TTCN-3 syntax BNF productions

This clause includes all BNF productions that are modifications of BNF rules defined in the TTCN-3 core language document ETSI ES 201 873-1 [1]. When using this package the BNF rules below replace the corresponding BNF rules in the TTCN-3 core language document. The rule numbers define the correspondence of BNF rules.

### BNF changes in clause A.1.6.1.1 Type definitions

```
12. StructuredTypeDef ::= RecordDef |
                        UnionDef |
                        SetDef |
                        RecordOfDef |
                        SetOfDef |
                        EnumDef |
                        PortDef |
                        ComponentDef |
                        ClassDef
```

### BNF changes in clause A.1.6.1.4 Function definitions

```
158. FunctionDef ::= FunctionKeyword [ DeterministicModifier ]
                   Identifier "(" [ FunctionFormalParList ] ")"
                   [ RunsOnSpec ] [ MtcSpec ] [ SystemSpec ] [ ReturnType ] [ ExceptionSpec ]
                   StatementBlock
169. StatementBlock ::= BasicStatementBlock [ CatchBlocks ] [ FinallyBlock ]
176. FunctionRef ::= [ ( Identifier | ObjectInstance ) Dot ]
                   ( Identifier | PreDefFunctionIdentifier )
```

### BNF changes in clause A.1.6.1.6 Testcase definitions

```
185. TestcaseDef ::= TestcaseKeyword Identifier "(" [ TemplateOrValueFormalParList ] ")" ConfigSpec
                   StatementBlock
```

### BNF changes in clause A.1.6.1.7 Altstep definitions

```
192. AltstepDef ::= AltstepKeyword
                   Identifier "(" [ FunctionFormalParList ] ")"
                   [ RunsOnSpec ] [ MtcSpec ] [ SystemSpec ] [ ExceptionSpec ]
                   "{ " AltstepLocalDefList AltGuardList " }" [ CatchBlocks ] [ FinallyBlock ]
```

### BNF changes in clause A.1.6.1.10 External function definitions

```
235. ExtFunctionDef ::= ExtKeyword FunctionKeyword [ DeterministicModifier ]
                       Identifier "(" [ FunctionFormalParList ] ")"
                       [ ReturnType ] [ ExceptionSpec ]
```

### BNF changes in clause A.1.6.4.1 Component Operations

```
261. ConfigurationOps ::= CreateOp |
                          SelfOp |
                          SystemKeyword |
                          MTCKeyword |
                          RunningOp |
                          AliveOp |
                          ThisOp |
                          SuperOp
```

### BNF changes in clause A.1.6.8.2 Behaviour statements

```
479. BehaviourStatements ::= TestcaseInstance |
                             FunctionInstance |
                             ReturnStatement |
                             AltConstruct |
                             InterleavedConstruct |
                             LabelStatement |
                             GotoStatement |
                             RepeatStatement |
                             DeactivateStatement |
                             AltstepInstance |
                             ActivateOp |
```

[BreakStatement](#) |  
[ContinueStatement](#) |  
[RaiseExceptionStatement](#)

### BNF changes in A.1.6.8.3 Basic statements

548. RelOp ::= "<" | ">" | ">=" | "<=" | [OfKeyword](#)

## A.3 Additional TTCN-3 syntax BNF productions

This clause includes all additional BNF productions that needed to define the syntax introduced by this package. All rules start with the digits "0330".

### Additional BNF rules related to clause A.1.6.1.1 Type definitions

```
033001. ClassDef ::= [ ExtKeyword ] ClassKeyword [ FinalModifier | AbstractModifier ]
    Identifier [ ExtendsKeyword Type ] [ RunsOnSpec ] [ MtcSpec ] [ SystemSpec ]
    "{" ClassMemberList ""
    [ FinallyKeyword BasicStatementBlock ]
033002. ClassKeyword ::= "class"
033003. ThisOp ::= "this"
033004. SuperOp ::= "super"
033005. FinalModifier ::= "@final"
033006. AbstractModifier ::= "@abstract"
033007. FinallyKeyword ::= "finally"
033008. ObjectType ::= "object"
033009. ClassMemberList ::= { ClassMember [ WithStatement ] [ SemiColon ] }
033010. ClassMember ::= [ MemberVisibility ]
    ( VarInstance |
      TimerInstance |
      ConstDef |
      TemplateDef |
      ClassFunctionDef |
      ConstructorDef )
033011. MemberVisibility ::= "public" | "private"
033012. ClassFunctionDef ::= [ ExtKeyword ] FunctionKeyword
    [ FinalModifier | AbstractModifier ] [ DeterministicModifier ]
    Identifier "(" [ FunctionFormalParList ] ")" [ ReturnType ]
    [ StatementBlock ]
033013. ConstructorDef ::= CreateKeyword
    "(" FunctionFormalParList ")" [ ":" FunctionInstance ]
    [ StatementBlock ]
```

### Additional BNF rules related to clause A.1.6.1.4 Function definitions

```
033014. BasicStatementBlock ::= "{" [ FunctionDefList ] [ FunctionStatementList ] ""
033015. CatchBlocks ::= CatchBlock { CatchBlock }
033016. CatchBlock ::= CatchOpKeyword "(" Type Identifier ")" BasicStatementBlock
033017. FinallyBlock ::= FinallyKeyword BasicStatementBlock
033018. ObjectInstance ::= ( ExtendedIdentifier | FunctionInstance ) [ ExtendedFieldReference ]
```

### Additional BNF rules related to clause A.1.6.8.2 Behaviour statements

```
033019. RaiseExceptionStatement ::= RaiseKeyword TemplateInstance
/* STATIC SEMANTICS - The TemplateInstance shall evaluate to an explicit value. */
```



---

## History

<b>Document history</b>		
V1.1.1	October 2018	Membership Approval Procedure MV 20181225: 2018-10-26 to 2018-12-25
V1.1.1	January 2019	Publication