ETSI ES 202 785 V1.9.1 (2022-05)

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

Reference
RES/MTS-202785v191

Keywords
conformance, testing, TTCN-3

ETSI

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

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

Siret N° 348 623 562 00017 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are 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 Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, 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.

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.

Foreword

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

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

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

Modal verbs terminology

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

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

1 Scope

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

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

This package defines types for behaviour definitions in 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 and guidelines for this package in combination with other packages is outside the scope of the present document.

2 References

2.1 Normative references

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

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

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

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

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

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

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

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

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

[6] ETSI ES 202 784: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Advanced Parameterization".

[7] 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".

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".

3 Definition of terms, symbols and abbreviations

3.1 Terms

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

behaviour definition: definition of an altstep, function, or testcase that can be called explicitly

NOTE: A control part is not considered a behaviour definition, because it cannot be called explicitly.

behaviour type: type of behaviour definitions

NOTE: Behaviour types are of kind altstep, function, or testcase.

3.2 Symbols

Void.

3.3 Abbreviations

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

4 Package conformance and compatibility

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

`"TTCN-3:2009 Behaviour Types"` - *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; in ETSI ES 201 873-1 [1] and ETSI ES 201 873-4 [2]. All features marked [AdvancedParameterization] have to be implemented only in case that this package is used together with the Advanced Parameterization package [6].

The package presented in the present document is compatible with:

- ETSI ES 201 873-1 [1] (V4.5.1).
- ETSI ES 201 873-4 [2] (V4.4.1).
- ETSI ES 201 873-5 [3] (V4.5.1).
- ETSI ES 201 873-6 [4] (V4.5.1).
- ETSI ES 201 873-7 [i.1] (V4.5.1).
- ETSI ES 201 873-8 [i.2] (V4.5.1).
- ETSI ES 201 873-9 [i.3] (V4.5.1).
- ETSI ES 201 873-10 [7] (V4.5.1).

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

The package presented in the present document is also compatible with:

- ETSI ES 202 784 [6] Package Advanced Parameterization (V1.3.1);

and can be used together with this package.

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

5 Package concepts for the core language

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

Clause 5.4 Parameterization

Values of behaviour types can be passed as parameters as indicated in table 2.

Table 2: Overview of parameterizable TTCN-3 objects

Keyword	Allowed kind of Parameterization	Allowed form of Parameterization	Allowed types in formal parameter lists
module	Value parameterization	Static at start of run-time	all basic types, all user-defined types and **`address`** type.
type (see note)	Value parameterization	Static at compile-time	all basic types, all user-defined types and **`address`** type.
template	Value and template parameterization	Dynamic at run-time	all basic types, all user-defined types, **`address`** type, **`template`**, and behaviour types.
function	Value, template, port and timer parameterization	Dynamic at run-time	all basic types, all user-defined types, **`address`** type, **`component`** type, **`port`** type, **`default`**, behaviour types, **`template`** and **`timer`**.
altstep	Value, template, port and timer parameterization	Dynamic at run-time	all basic types, all user-defined types, **`address`** type, **`component`** type, **`port`** type, **`default`**, behaviour types, **`template`** and **`timer`**.
testcase	Value, template, port and timer parameterization	Dynamic at run-time	all basic types and of all user-defined types, **`address`** type, **`template`**, and behaviour types.
signature	Value and template parameterization	Dynamic at run-time	all basic types, all user-defined types and **`address`** type, **`component`** type, and behaviour types.
NOTE: Record of, set of, enumerated, port, component and sub-type definitions do not allow parameterization.			

Clause 5.4.1.1 Formal parameters of kind value

Also, values of behaviour types can be passed as value parameters.

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

Behaviour types such as **`altstep`**, **`function`**, and **`testcase`** may be used to define flexible behaviour of TTCN-3 libraries.

No subtyping is defined for behaviour types.

Clause 6.2 Structured Types and Values

Extend clause 6.2 Structured types and values by the following clause 6.2.13.

6.2.13 Behaviour Types

6.2.13.1 Behaviour Type Definitions

Behaviour types are the set of identifiers of altstep, function, and testcase definitions with a specific parameter list, runs on, system, mtc and return clauses. They denote those altsteps, functions, and testcases, respectively, defined in the test suite that have a compatible parameter list and compatible **`runs on`** or **`system`** clauses.

Syntactical Structure

```
type  function  BehaviourTypeIdentifier
[ "<" { FormalTypePar [","] } ">" ] NOTE 1
"(" [ { ( FormalValuePar | FormalTimerPar | FormalTemplatePar | FormalPortPar ) [","] } ] ")"
[ runs on ( ComponentType | self ] NOTE 2
[ mtc ComponentType ]
[ system ComponentType ]
[ return [ template ] Type ]

type altstep BehaviourTypeIdentifier
[ "<" { FormalTypePar [","] } ">" ] NOTE 1
"(" [ { ( FormalValuePar | FormalTimerPar | FormalTemplatePar | FormalPortPar ) [","] } ] ")"
[ runs on ( ComponentType | self ] NOTE 2
[ mtc ComponentType ]
[ system ComponentType ]

type testcase BehaviourTypeIdentifier
[ "<" { FormalTypePar [","] } ">" ] NOTE 1
"(" [ { ( FormalValuePar | FormalTemplatePar ) [","] } ] ")"
runs on ComponentType
[ system ComponentType ]
```

Semantic Description

Behaviour types define prototypes of altsteps, functions, and testcases.

NOTE 1: [AdvancedParameterization] If the advanced parameterization package [6] is also supported, behaviour types can have type parameters.

NOTE 2: **`runs on self`** indicates a specific compatibility check, see extension of clause 6.3.

Restrictions

a) The rules for formal parameter lists shall be followed as defined in the TTCN-3 Core Language [1] clause 5.4 and extended in clause 5.1 of the present document.

b) Behaviour types of kind **`altstep`** may have a **`runs on`**, **`system`** or **`mtc`** clause, behaviour types of type **`function`** may have a **`runs on`**, **`system`**, **`mtc`** or **`return`** clause, behaviour types of kind **`testcase`** shall have a **`runs on`** clause and may have a **`system`** clause.

c) **`runs on self`** shall not be used for test cases.

Examples

EXAMPLE 1: Function type with one parameter and a return value.

```
type function MyFunc1 ( in integer p1 ) return boolean;
```

EXAMPLE 2: Function type with one parameter, a **runs on** clause and a return value.

```
type function MyFunc2 ( in integer p1 ) runs on MyCompType return boolean;
```

EXAMPLE 3: Altstep type with a type parameter and a **runs on** clause.

```
type altstep MyAltstep1<type T> ( in T p1 ) runs on MyCompType;
```

EXAMPLE 4: Testcase type without parameter, with a **runs on** clause and a **system** clause.

```
type testcase MyTestcase1 ( ) runs on MyCompType system MySysType;
```

6.2.13.2 Behaviour Values

The values of a behaviour type are the identifiers of altsteps, functions, and testcases with compatible parameters and **runs on, system**, **mtc** and **return** clauses. Both predefined and user-defined, including external, functions can be used as values. Type compatibility of behaviour types is defined in the extension to clause 6.3.5 within the present document.

Syntactical Structure

VariableRef | *FunctionInstance* | *FunctionRef* | *AltstepRef* | *TestcaseRef* | null

Semantic Description

The literal behaviour values are the identifiers of the predefined and user-defined altsteps, (external) functions, and testcases and the special value **null**. The special value **null** is available to indicate an undefined behaviour value, e.g. for the initialization of variables. Behaviour values can be passed around as parameters and behaviour values can be stored. Behaviour values can be used, together with a corresponding list of actual parameters, to invoke the behaviours in statements and expressions. Behaviour values can also be used, again together with a corresponding list of actual parameters, in **activate**, **start**, and **execute** statements, respectively.

The only operators (see clause 7.1 of the TTCN-3 core language [1]) on behaviour values that are defined are the check for equality and inequality.

Restrictions

a) Values of a behaviour type with a **runs on self** clause shall not be sent to another test component.

b) Values of a behaviour type with a **runs on self** clause shall not be used in a start test component operation.

c) The special value **null** shall not be used to invoke a behaviour.

Examples

```
type function MyFunc3 ( in integer p1 ) return charstring;
function blanks (in integer p1) return charstring {
    // return a charstring of p1 blank characters
}
var MyFunc3 myVar1 := blanks;
var MyFunc3 myVar1 := int2char;
```

6.2.13.3 Deferred Behaviour Type Definitions

Deferred behaviour types are the set of altstep, function, and testcase behaviours with possible runs on, system and mtc clauses paired with their actual parameter list which can be passed to the **activate**, **start** and **execute** operations, respectively. They denote those instantiations of altsteps, functions, and testcases defined in the test suite that have compatible **`runs on`**, **`system`** or **`mtc`** clauses.

Syntactical Structure

```
type  function  BehaviourTypeIdentifier
[ runs on ComponentType ]
[ mtc ComponentType ]
[ system ComponentType ]

type altstep BehaviourTypeIdentifier
[ runs on ComponentType ]
[ mtc ComponentType ]
[ system ComponentType ]

type testcase BehaviourTypeIdentifier
runs on ComponentType
[ system ComponentType ]
```

Semantic Description

Deferred behaviour types define references to executable behaviours. Deferred function behaviour types define the set of behaviours that can be started on a ptc. Deferred altstep behaviour types define the set of behaviours that can be activated as default alternatives. Deferred testcase behaviour types define the set of behaviours that can be executed from the control part.

The actual parameters of a deferred behaviour value are evaluated when creating the value.

Restrictions

a) The same restrictions apply for the start, activate and execute operations that apply for functions, altsteps and testcases used directly in these statements in a non-deferred way.

Examples

EXAMPLE 1: Deferred function type with.

```
type function MyFunc1;
```

EXAMPLE 2: Deferred function type with a **runs on** clause.

```
type function MyFunc2 runs on MyCompType;
```

EXAMPLE 3: Deferred altstep type with a **runs on** clause.

```
type altstep MyAltstep1 runs on MyCompType;
```

EXAMPLE 4: Deferred testcase type with a **runs on** clause and a **system** clause.

```
type testcase MyTestcase1 runs on MyCompType system MySysType;
```

6.2.13.4 Deferred Behaviour Values

The values of a deferred behaviour type are the un-executed instantiations of altsteps, functions, and testcases together with their compatible actual parameters which have compatible runs on, system and mtc clauses to those in the behaviour type. Type compatibility of deferred behaviour types is defined in the extension to clause 6.3.5 within the present document.

Syntactical Structure

```
VariableRef | FunctionInstance
```

Semantic Description

The literal deferred behaviour values are the function instance expressions of the user-defined altsteps, (external) functions, and testcases. Deferred behaviour values can be passed around as actual parameters or stored in a variable. Deferred altstep, function or testcase behaviour values can be used in **`activate`**, **`start`**, and **`execute`** statements, respectively.

The only operators (see clause 7.1 of the TTCN-3 core language [1]) on deferred behaviour values that are defined are the check for equality and inequality.

Restrictions

a) Only in functions, altsteps and testcases without inout or out parameters shall be used as deferred behaviour values.

b) For the actual parameters of a function behaviour value instance, restrictions from clause 21.3.2 of the core language specification shall apply.

c) For the actual parameters of an alstep behaviour value instance, restrictions from clause 20.5.2 of the core language specification shall apply.

d) For the actual parameters of a testcase behaviour value instance, restriction from clause 26.1 of the core language specification shall apply.

Examples

EXAMPLE 1:

```
type function MyFunctionBehaviour;
function MyFunction (in integer p1) {
    // …
}
var MyFunctionBehaviour v_myBehaviour := MyFunction(4); // does not invoke MyFunction(4)
var PtcType ptc := PtcType.create;
ptc.start(myBehaviour); // starts MyFunction(4) on component ptc
```

EXAMPLE 2:

```
type altstep MyAltstepType runs on PtcType;
altstep MyAltstep(integer i) runs on PtcType {
[] …
}
var MyAltstepType v_myAltstep := MyAltstep(3);
activate(v_myAltstep);
```

EXAMPLE 3:

```
type testcase MyTestcaseBehaviour runs on PtcType;
testcase MyTestcase(integer i) runs on PtcType { … }
function runTestCase(MyTestcaseBehaviour p_myTestcase) {
    execute(p_myTestcase, 1.0);
}
control {
    var MyTestcaseBehaviour v_myTestcase := MyTestcase(3);
    runTestCase(v_myTestcase);
    runTestCase(MyTestcase(4)); // execution is deferred to runTestCase
}
```

Clause 6.3 Type compatibility

Clause 6.3 Type compatibility is extended by:

6.3.5 Type compatibility of behaviour types

Altsteps are only compatible to **altstep** behaviour types, functions are only compatible to **function** behaviour types, testcases are only compatible to **testcase** behaviour types. A behaviour object (an altstep, function or testcase) is a value of a given behaviour type, if the parameter lists are compatible, if the return clause is compatible, if the **runs on**, **mtc** and **system** clauses are compatible, provided they exist, and if modifiers (optionally) declared for the behaviour object and the given behaviour type are identical.

The parameter list of an object is compatible with the parameter list of a type if the order of the parameters is identical, as well as direction, kind, type, name of the parameters, optional modifiers and whether a default exists. If the parameter is of kind **template**, then also potential template restrictions have to be identical. Compatibility of parameter lists applies to the type parameter list, if exists (i.e. when the advanced parameterization package [6] is also supported and a type parameter list is defined), and the value parameter list separately.

The return clause of a function is compatible with the return clause of a function type if it is either absent in case the function type does not have a return clause, or it is of identical kind and type if the function type has a return clause. In case the return clause is of kind **template**, then potential template restrictions have to be identical, too.

The **runs on** clause of an object is compatible with the **runs on** clause of a behaviour type, if it is either absent in the object or, if the **runs on** clause exists, the component type in the **runs on** clause of the object is runs on compatible (see ETSI ES 201 873-1 [1], clause 6.3.3, section 2)) with the type specified in the **runs on** clause of the behaviour type. According to the first condition, an object without a **runs on** clause is compatible with a behaviour type that has a **runs on** clause.

The **system** clause of a **testcase** object is compatible with the **system** clause of a **testcase** type, if the component type in the **system** clause of the **testcase** object is system compatible (see ETSI ES 201 873-1 [1], clause 6.3.3, section 4)) with the type specified in the **system** clause of the **testcase** type. If the **testcase** object or the **testcase** behaviour type does not have a **system** clause, then the component type of the **runs on** clause is used instead (see ETSI ES 201 873-1 [1], clause 9.2).

The **system** clause of a function or altstep object is compatible with the **system** clause of a function or altstep behaviour type, if it is either absent in the function or altstep object or, if the **system** clause exists, the component type in the **system** clause of the function or altstep object is system compatible (see ETSI ES 201 873-1 [1], clause 6.3.3, section 4)) with the type specified in the **system** clause of the function or altstep behaviour. According to the first condition, a function or altstep object without a **system** clause is compatible with a function or altstep behaviour type that has a **system** clause.

The **mtc** clause of an object is compatible with the **mtc** clause of a behaviour type, if it is either absent in the object or, if the **mtc** clause exists, the component type in the **mtc** clause of the object is mtc compatible (see ETSI ES 201 873-1 [1], clause 6.3.3, section 3)) with the type specified in the **mtc** clause of the behaviour type. According to the first condition, an object without an **mtc** clause is compatible with a behaviour type that has an **mtc** clause.

For functions it does not matter whether the function is an external function. If the parameter lists are compatible, then an external function is also compatible with the function behaviour type.

EXAMPLE:

```
// Given
type component MyCompType0 { };
type component MyCompType1 { integer a };
type component MyCompType2 { integer a;
                             float   b };
type function MyFunc ( in integer p1 ) runs on MyCompType1 return boolean;
type function MyFuncNoRunsOn ( in integer p1 ) return boolean;

//compatible with MyFunc, identical parameterlist, runs on clause, return type
function f1 (in integer p1) runs on MyCompType1 return boolean { /*…*/ };
//compatible with MyFunc, component type in type extends the one of the function
function f2 (in integer p1) runs on MyCompType0 return boolean { /* … */ }
//compatible with MyFunc, function does not have runs on clause
function f3 (in integer p1) return boolean { /* … */ }
```

```
//incompatible with MyFunc, additional parameter without default value
function g1 (in integer p1, in boolean p2) runs on MyCompType1 return boolean { /*…*/ };
//incompatible with MyFunc, missing return clause
function g2 (in integer p1) runs on MyCompType1 { /*…*/ };
//incompatible with MyFunc, different kind of parameter
function g3 (in template integer p1) runs on MyCompType1 return boolean { /*…*/ };
//incompatible with MyFunc, component type of function is an extension of the function type
function g4 (in integer p1) runs on MyCompType2 return boolean { /* … */ }
//incompatible with MyFuncNoRunsOn, function has runs on clause, type does not have one
function g5 (in integer p1) runs on MyCompType1 return boolean { /* … */ }
```

6.3.6 Type compatibility of behaviour types with runs on self

Function and altstep types can be defined with a **runs on self** clause. If a specific value is assigned to a variable or parameter of such a behaviour type, then the **runs on** clause of the enclosing definition is used in the compatibility check.

A value of a behaviour type with a **runs on self** clause can always be used in a function or altstep invocation.

Restriction

a) Behaviour type definitions of module parameters and global constants shall have no **runs on self** clause.

NOTE: The compatibility check is not possible for module parameters and global constants defined in the module definitions because the module definitions part has no **runs on** clause.

EXAMPLE:

```
// Given
type component MyCompType0 { };
type component MyCompType1 { integer a };
type component MyCompType2 { integer a;
                             float   b };
type function MyFuncType ( in integer p1 ) runs on self;

function MyFunc1 ( in integer p1 ) runs on MyCompType1 { /* … */ };
function MyFunc2 () runs on MyCompType2 {
  var MyFuncType func;
  func := MyFunc1;  // correct,
                    // MyCompType1 of MyFunc1 is compared against MyCompType2 of MyFunc2
:
}
```

6.3.7 Type compatibility of deferred behaviour types

Deferred altstep behaviour values are only compatible to deferred **altstep** behaviour types, deferred function behaviour values are only compatible to deferred **function** behaviour types, deferred testcase behaviour values are only compatible to deferred **testcase** behaviour types. A behaviour object (an altstep, function or testcase) is a value of a given behaviour type, if the **runs on**, **system** and mtc clauses are compatible, provided they exist.

The **runs on** clause of a deferred object is compatible with the **runs on** clause of a deferred behaviour type, if it is either absent in the object or, if the **runs on** clause exists, the component type in the **runs on** clause of the object is runs on compatible (see ETSI ES 201 873-1 [1], clause 6.3.3, condition 2)) with the type specified in the **runs on** clause of the deferred behaviour. According to the first condition, an object without a **runs on** clause is compatible with a deferred behaviour type that has a **runs on** clause.

The **system** clause of a deferred **testcase** behaviour object is compatible with the **system** clause of a deferred **testcase** behaviour type, if the component type in the **system** clause of the **testcase** object is system compatible (see ETSI ES 201 873-1 [1], clause 6.3.3, condition 4)) with the type specified in the **system** clause of the deferred **testcase** behaviour type. If the **testcase** object or the deferred **testcase** behaviour type does not have a **system** clause, then the component type of the **runs on** clause is used instead (see ETSI ES 201 873-1 [1], clause 9.2).

The **system** clause of a deferred function or altstep object is compatible with the **system** clause of a deferred function or altstep behaviour type, if it is either absent in the function or altstep object or, if the **system** clause exists, the component type in the **system** clause of the function or altstep object is system compatible (see ETSI ES 201 873-1 [1], clause 6.3.3, condition 4)) with the type specified in the **system** clause of the deferred function or altstep behaviour type. According to the first condition, a function or altstep object without a **system** clause is compatible with a deferred function or altstep behaviour type that has a **system** clause.

The **mtc** clause of a deferred object is compatible with the **mtc** clause of a deferred behaviour type, if it is either absent in the object or, if the **mtc** clause exists, the component type in the **mtc** clause of the object is mtc compatible (see ETSI ES 201 873-1 [1], clause 6.3.3, condition 3)) with the type specified in the **mtc** clause of the deferred behaviour type. According to the first condition, an object without an **mtc** clause is compatible with a deferred behaviour type that has an **mtc** clause.

EXAMPLE:

```
// Given
type component MyCompType0 { };
type component MyCompType1 { integer a };
type component MyCompType2 { integer a;
                             float   b };
type function MyFunc runs on MyCompType1;
type function MyFuncNoRunsOn;

// instantiations of f1 are compatible with MyFunc runs on clause
function f1 (in integer p1) runs on MyCompType1 return boolean { /*…*/ };
// instantiations of f2 are compatible with MyFunc, component type in type
// extends the one of the function
function f2 (in integer p1) runs on MyCompType0 return boolean { /* … */ }
// instantiations of f3 compatible with MyFunc, function does not have runs on clause
function f3 (in integer p1) return boolean { /* … */ }

//incompatible with MyFunc, component type of function is an extension of the function type
    function g4 (in integer p1) runs on MyCompType2 return boolean { /* … */ }
//incompatible with MyFuncNoRunsOn, function has runs on clause, type does not have one
    function g5 (in integer p1) runs on MyCompType1 return boolean { /* … */ }
```

5.3 Extension to ETSI ES 201 873-1, clause 7 (Expressions)

Values of behaviour types can be passed around as parameters, stored in variables, exchanged among components, compared against each other, and applied to parameter lists. No other operation is defined on values of behaviour types.

Values of deferred behaviour types can be passed around as parameters, stored in variables, exchanged among components, compared against each other, and used in the component start, activate and execute operations, dependant on their behaviour kind. No other operation is defined on values of behaviour types.

Clause 7.1.3 Relational operators

The operators of equality and non-equality can be applied to values of the same behaviour type. Each behaviour value can be compared with **null**. No other value is equal to **null**.

The operators of equality and non-equality can be applied to values of the same deferred behaviour type.

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

Clause 8.2.1 Module parameters

Module parameters of behaviour types can be declared.

Module parameters of deferred behaviour types can be declared.

5.5 Extension to ETSI ES 201 873-1, clause 10 (Declaring constants)

Constants of behaviour types can be declared.

Constants of deferred behaviour types can be declared.

5.6 Extension to ETSI ES 201 873-1, clause 11 (Declaring variables)

Variables of behaviour types can be declared.

Variables of deferred behaviour types can be declared.

5.7 Extension to ETSI ES 201 873-1, clause 15 (Declaring templates)

Clause 15.7 Template matching mechanisms

Template matching mechanisms can be used for behaviour types and deferred behaviour types as indicated in the following extension of table 11 TTCN-3 matching mechanisms.

Used with values of	**Value**		**Instead of values**								**Inside values**			**Attributes**	
	SpecificValue	OmitValue	ComplementedList	ValueList	AnyValue (?)	AnyValueOrNone (see note)	Range	Superset	Subset	Pattern	Anyelement (?)	AnyElementsOrNone (see note)	Permutation	LengthRestriction	IfPresent
altstep	Yes	Yes	Yes	Yes	Yes	Yes (see note 3)									Yes (see note 3)
function	Yes	Yes	Yes	Yes	Yes	Yes (see note 3)									Yes (see note 3)
testcase	Yes	Yes	Yes	Yes	Yes	Yes (see note 3)									Yes (see note 3)
...															
NOTE 3: When used, shall be applied to optional fields of record and set types only (without restriction on the type of that field).															

5.8 Extension to ETSI ES 201 873-1, clause 16 (Functions, altsteps and test cases)

Clause 16.1.1 Invoking functions

A function can also be invoked by the **apply** operation by referring to an expression of a behaviour type of kind function and to a parameter list.

Syntactical Structure

```
apply "(" Value "(" [ { ( TimerRef | TemplateInstance |
                          Port | ComponentRef | "-" ) [","] } ] ")" ")"
```

NOTE: This syntactical structure supplements the syntactical structure in ETSI ES 201 873-1 [1] with an alternative option.

Restrictions

g) The value in an **apply** operation shall be an expression of a function type but it shall not be the literal name of a function.

Examples

```
var MyFuncType v_func := …;
apply(v_func(MyVar2));          // the function stored in v_func is applied to MyVar2
```

Clause 16.2.1 Invoking altsteps

An altstep can be invoked by referring to a value of a behaviour type of kind altstep.

Syntactical Structure

```
apply "(" Value "(" [ { ( TimerRef | TemplateInstance |
                          Port | ComponentRef | "-" ) [","] } ] ")" ")"
```

NOTE: This syntactical structure supplements the syntactical structure in ETSI ES 201 873-1 [1] with an alternative option.

Restrictions

a) The value in an **apply** operation shall be an expression of an altstep type but it shall not be the literal name of an altstep.

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

Clause 19.11 The log statement

Values of behaviour types shall be logged as indicated in the following extension of table 18 TTCN-3 language elements that can be logged.

Used in a log statement	What is logged	Comment
behaviour type variable identifier	actual value or "UNINITIALIZED"	See notes 4 and 9.
formal parameter identifier	see comment column	In case of behaviour type parameters the actual value shall be logged (see notes 4 and 9).
NOTE 4: The string "UNINITIALIZED" is logged only if the log item is unbound (uninitialized). NOTE 9: For values of behaviour types the name of the definition qualified with the module of its definition shall be logged. For predefined functions the function name only shall be logged.		

Values of deferred behaviour types shall be logged as indicated in the following extension of table 18 TTCN-3 language elements that can be logged.

Used in a log statement	What is logged	Comment
Deferred behaviour type variable identifier	actual value or "UNINITIALIZED"	See notes 4 and 10.
formal parameter identifier	see comment column	In case of deferred behaviour type parameters the actual value shall be logged (see notes 4 and 10).
NOTE 4: The string "UNINITIALIZED" is logged only if the log item is unbound (uninitialized). NOTE 10: For values of deferred behaviour types the name of the definition qualified with the module of its definition shall be logged. For predefined functions the function name only shall be logged. In both cases, additionally, the actual parameter lists of the behaviour value are logged.		

5.10 Extension to ETSI ES 201 873-1, clause 20 (Statement and operations for alternative behaviours)

Clause 20.5.2 The activate operation

Altsteps can also be activated by referring to a value of an altstep type.

Syntactical Structure

```
activate "(" apply "(" Value "(" [ { ( TimerRef | TemplateInstance |
                                       Port | ComponentRef | "-" ) [","] } ] ")" ")" ")"
```

NOTE 1: This syntactical structure supplements the syntactical structure in ETSI ES 201 873-1 [1] with an alternative option.

Restrictions

a) The value in the **`apply`** operation shall be an expression of an altstep type but it shall not be the literal name of an altstep.

Altsteps can also be activated by referring to a deferred altstep behaviour value.

Syntactical Structure

```
activate "(" VariableRef ")"
```

NOTE 2: This syntactical structure supplements the syntactical structure in ETSI ES 201 873-1 [1] with an alternative option.

Restrictions

a) VariableRef shall refer to a deferred altstep behaviour.

5.11 Extension to ETSI ES 201 873-1, clause 21 (Configuration Operations)

Clause 21.2.2 The start test component operation

Functions can also be started on other components by referring to a value of a function type.

Syntactical Structure

```
( VariableRef | FunctionInstance ) "." start "(" apply "(" Value "("
                      [ { ( TimerRef | TemplateInstance | Port | ComponentRef | "-" ) [","] } ]
                      ")" ")""")"
```

NOTE 1: This syntactical structure supplements the syntactical structure in ETSI ES 201 873-1 [1] with an alternative option.

Restrictions

a) The value in the **apply** operation shall be an expression of a function type but it shall not be the literal name of a function.

Examples

```
function MyFirstBehaviour() runs on MyComponentType { … }
type function MyBehaviourType () runs on MyComponentType
:
var MyComponentType MyNewPTC;
var MyComponentType MyAlivePTC;
var MyBehaviourType MyFunction := MyFirstBehaviour;
:
MyNewPTC := MyComponentType.create;         // Creation of a new non-alive test component.
MyAlivePTC := MyComponentType.create alive; // Creation of a new alive-type test component
:
MyAlivePTC.start(apply(MyFunction()));    // Start function indicated by variable MyFunction
```

Functions can also be started on other components by referring to a deferred function behaviour value.

Syntactical Structure

```
( VariableRef | FunctionInstance ) "." start "(" VariableRef ")"
```

NOTE 2: This syntactical structure supplements the syntactical structure in ETSI ES 201 873-1 [1] with an alternative option.

Restrictions

a) The *VariableRef* given as parameter to the start operation shall refer to a deferred function behaviour.

Examples

```
function MyFirstBehaviour() runs on MyComponentType { … }
type function MyBehaviourType runs on MyComponentType
:
var MyComponentType MyNewPTC;
var MyComponentType MyAlivePTC;
var MyBehaviourType MyFunction := MyFirstBehaviour();

:
MyNewPTC := MyComponentType.create;         // Creation of a new non-alive test component.
MyAlivePTC := MyComponentType.create alive; // Creation of a new alive-type test component
:
MyAlivePTC.start(MyFunction);       // Start function indicated by variable MyFunction
```

5.12 Extension to ETSI ES 201 873-1, clause 26 (Module control)

Clause 26.1 The execute statement

A testcase can also be executed by referring to a value of a testcase type.

Syntactical Structure

This syntactical structure supplements the syntactical structure in ETSI ES 201 873-1 [1] with an alternative option:

```
execute "(" apply "(" Value "(" [ { ( TemplateInstance | "-" ) ["("] } ] ")"
            [ "," TimerValue ] ")" ")"
```

Restrictions

a) The value in the **apply** operation shall be an expression of a testcase type but it shall not be the literal name of a testcase.

A testcase can also be executed by referring to a deferred testcase behaviour value.

Syntactical Structure

This syntactical structure supplements the syntactical structure in ETSI ES 201 873-1 [1] with an alternative option:

```
execute "(" VariableRef
            [ "," TimerValue ] ")" ")"
```

Restrictions

a) The *VariableRef* given as parameter to the **execute** operation shall refer to a deferred testcase behaviour.

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

5.13.0 New keywords and TTCN-3 syntax BNF productions

New TTCN-3 Keywords

The list of keywords is extended by the following keywords:

apply

New TTCN-3 syntax BNF productions

The BNF is extended with the following productions.

NOTE: [AdvancedParameterization] FormalTypeParList is applicable only when the advanced parameterization package [6] is also supported.

```
785001. BehaviourDef ::= ( AltstepKeyword TypeReference [FormalTypeParList]
                           [ "("[FunctionFormalParList] ")" ] [RunsOnSpec] [MtcSpec] [SystemSpec] ) |
                     ( FunctionKeyword TypeReference [FormalTypeParList]
                      [ "("[FunctionFormalParList] ")" ] [RunsOnSpec] [MtcSpec] [SystemSpec]
                      [ReturnType] ) |
                     ( TestcaseKeyword TypeReference [FormalTypeParList]
                      [ "("[TemplateOrValueFormalParList] ")"] ConfigSpec_StatementBlock)
785002. NestedBehaviourDef ::= ( AltstepKeyword "("[FunctionFormalParList] ")" [RunsOnSpec]
                     [MtcSpec] [SystemSpec] ) |
                     ( FunctionKeyword "("[FunctionFormalParList] ")" [RunsOnSpec] [ReturnType]
                     [MtcSpec] [SystemSpec] ) |
                     ( TestcaseKeyword "("[TemplateOrValueFormalParList] ")"
                      ConfigSpec_StatementBlock)
785003. BehaviourValue ::= ExtendedIdentifier /*AltstepRef, FunctionRef or TestcaseRef */ | "null"
785004. ApplyKeyword ::= "apply"
```

5.13.1 Changes to ETSI ES 201 873-1, clause A.1.6 (TTCN-3 syntax BNF productions)

The following productions from [1] TTCN-3 Core Language clause A.1.6 are modified.

```
12. StructuredTypeDef ::= RecordDef | UnionDef | SetDef | RecordOfDef | SetOfDef | EnumDef |
                          PortDef | ComponentDef | MapDef | BehaviourDef
18. NestedTypeDef ::= NestedRecordDef | NestedUnionDef | NestedSetDef |
                      NestedRecordOfDef | NestedSetOfDef | NestedEnumDef | NestedBehaviourDef
167. RunsOnSpec ::= RunsKeyword OnKeyword ( ComponentType | SelfOp )
178. FunctionInstance ::= ( FunctionRef [ "(" [ActualParList] ")" ] ) |
                          ( ApplyKeyword "(" Primary "(" [ActualParList] ")" ")" ) |
                     ( ExtendedIdentifier )
/* STATIC SEMANTICS – the part is only optional if the FunctionRef uses the ControlKeyword and the
referenced control function has no formal parameters */
193. TestcaseInstance ::= ExecuteKeyword "(" ( ExtendedIdentifier ["(" [ActualParList] ")"] ) |
                          ( ApplyKeyword "(" Primary "(" [TestcaseActualParList] ")" ")" )
                          ["," (Expression | Minus) ["," SingleExpression]] ")"
424. PredefinedValue ::= Bstring |
                         BooleanValue |
                         CharStringValue |
                         Number |  /* IntegerValue */
```

```
Ostring |
Hstring |
VerdictTypeValue |
FloatValue |
AddressValue |
OmitKeyword |
BehaviourValue
```

6 Package semantics

6.1 Replacements

The following clauses shall replace the following clauses in ETSI ES 201 873-4 [2] (V4.4.1):

- Clause 6.2 replaces clause 9.2 Activate statement in ETSI ES 201 873-4 [2] (V4.4.1).
- Clause 6.3.0 replaces the main text part of clause 9.17 Execute statement in ETSI ES 201 873-4 [2] (V4.4.1).
- Clause 6.3.1 replaces clause 9.17.1 Flow graph segment <execute-without-timeout> in ETSI ES 201 873-4 [2] (V4.4.1).
- Clause 6.4.0 replaces the main text part of clause 9.24 Function call in ETSI ES 201 873-4 [2] (V4.4.1).
- Clause 6.4.1 replaces clause 9.24.4 Flow graph segment <user-def-func-call> in ETSI ES 201 873-4 [2] (V4.4.1).
- Clause 6.4.2 replaces clause 9.24.5 Flow graph segment <predef-ext-func-call> in ETSI ES 201 873-4 [2] (V4.4.1).
- Clause 6.5 replaces clause 9.46 Start component operation in ETSI ES 201 873-4 [2] (V4.4.1).

In the clauses 6.2 to 6.5 all figure numbers, figure references and clause references are related to the clauses and figures in ETSI ES 201 873-4 [2] (V4.4.1) and not to any clause or figure in the present document.

6.2 Activate statement

The syntactical structure of the **`activate`** statement is:

`activate`(<altstep-name>([<act-par-desc$_1$>, … , <act-par-desc$_n$>])) |
`activate`(**`apply`**(<altstep-expr>([<act-par-desc$_1$>, … , <act-par-desc$_n$>]))) |
`activate`(<deferred-altstep-behaviour>)

The <altstep-name> and the <altstep-expr> denote the identifier of the altstep that is activated as default behaviour, and <act-par-descr$_1$>, … , <act-par-descr$_n$> describe the actual parameter values of the altstep at the time of its activation. The <deferred-altstep-behaviour> denotes a reference to a deferred altstep behaviour. The operational semantics assumes that <altstep-name> is a literal name of the altstep type and can be handled like the <altstep-expr>, i.e. as an expression that evaluates to a value of the altstep behaviour type. Furthermore, it is assumed that for each <act-par-desc$_1$> the corresponding formal parameter identifier <f-par-Id$_1$> is known, i.e. the syntactical structure above can be extended to:

`activate`(<altstep-name>((<f-par-Id$_1$>,<act-par-desc$_1$>), … , (<f-par-Id$_n$>,<act-par-desc$_n$>))) |
`activate`(**`apply`**(<altstep-expr> ((<f-par-Id$_1$>,<act-par-desc$_1$>), … , (<f-par-Id$_n$>,<act-par-desc$_n$>))))

The flow graph segment <activate-stmt> in figure 37 defines the execution of the **`activate`** statement. The execution is structured into four steps. In the first step, the identifier of the altstep to be activated is determined. The identifier may directly be given by an altstep name or be given in form of an expression that evaluates to the altstep identifier. In the second step, a call record for the altstep is created. In the third step the values of the actual parameter are calculated and assigned to the corresponding field in the call record. In the fourth step, the call record is put as first element in the *DEFAULT-LIST* of the entity that activates the default.

NOTE: For altsteps that are activated as default behaviour, only value parameters are allowed. In figure 37, the handling of the value parameters is described by the flow graph segment <value-par-calculation>, which is defined in clause 9.24.1.

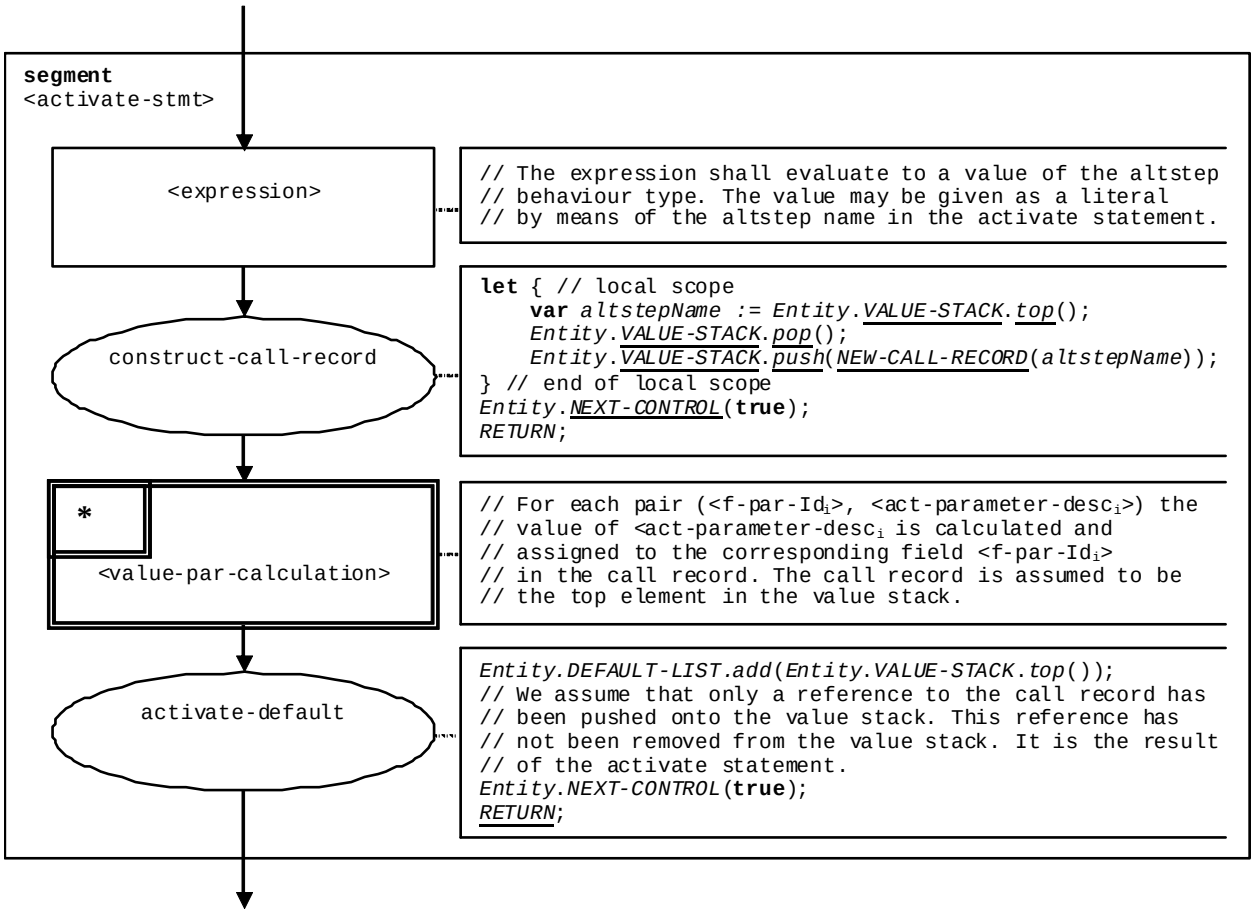

Figure 37: Flow graph segment <activate-stmt>

6.3 Replacements in Execute statement's operational semantics

6.3.0 Execute statement

The syntactical structure of the **execute** statement is:

execute(<testCaseId>([<act-par$_1$>, … , <act-par$_n$>)]) [, <float-expression>]) |
execute(**apply**(<testCase-expr>([<act-par$_1$>, … , <act-par$_n$>)]) [, <float-expression>])) |
execute(<deferred-testCase-behaviour> [, <float-expression>])

The **execute** statement describes the execution of a test case that is identified by a <testCaseId> or a <testCase-expr> with the (optional) actual parameters <act-par$_1$>, … , <act-par$_n$> or a <deferred-testcase-behaviour> which denotes a reference to a deferred testcase behaviour. Optionally the execute statement may be guarded by a duration provided in form of an expression that evaluates to a **float**. If within the specified duration the test case does not return a verdict, a timeout exception occurs, the test case is stopped and an **error** verdict is returned.

NOTE: The operational semantics models the stopping of the test case by a stop of the MTC. In reality, other mechanisms may be more appropriate.

If no timeout exception occurs, the MTC is created, the control instance (representing the control part of the TTCN-3 module) is blocked until the test case terminates, and for the further test case execution the flow of control is given to the MTC. The flow of control is given back to the control instance when the MTC terminates.

The flow graph segment <execute-stmt> in figure 67 defines the execution of an **execute** statement.

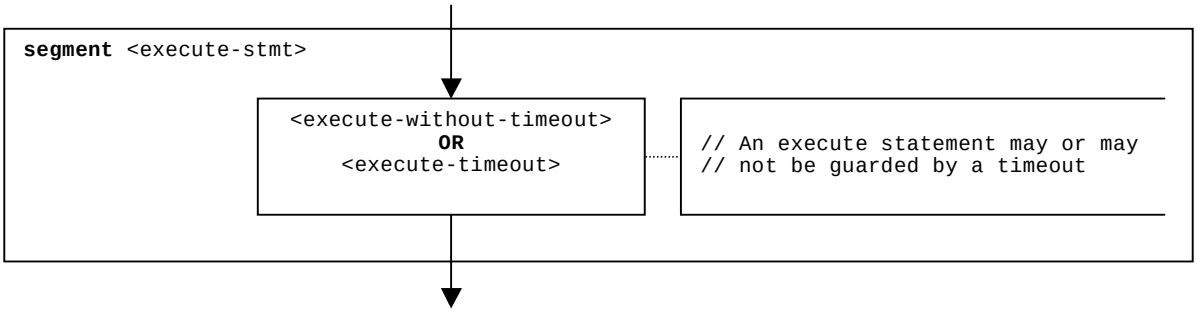

Figure 67: Flow graph segment <execute-stmt>

6.3.1 Flow graph segment <execute-without-timeout>

The execution of a test case starts with the creation of the **mtc**. Then the mtc is started with the behaviour defined in the test case definition. Afterwards, the module control waits until the test case terminates. The creation and the start of the MTC can be described by using **create** and **start** statements:

```
var MtcType myMTC := MtcType.create;
myMTC.start(TestCaseName(P1…Pn));
```

or, if the testcase is identified by a behaviour expression:

```
var MtcType myMTC := MtcType.create;
myMTC.start(apply(myTestCaseVar (P1…Pn)));
```

NOTE 1: For the replacements sketched above, the operational semantics assumes that all referenced definitions (e.g. MtcType, TestCaseName, myTestCaseVar) exist and that all definitions which are needed for the referenced definitions also exist (e.g. a testcase behaviour type definition for myTestCaseVar).

NOTE 2: For the case where the test case is identified by a behaviour expression, the operational semantics assumes for the replacement sketched above that the type of the MTC MtcType is referenced in the **runs on** clause of the behaviour type definition of variable myTestCaseVar.

The flow graph segment <execute-without-timeout> in figure 68 defines the execution of an **execute** statement without the occurrence of a timeout exception by using the flow graph segments of the operations **create** and the **start**.

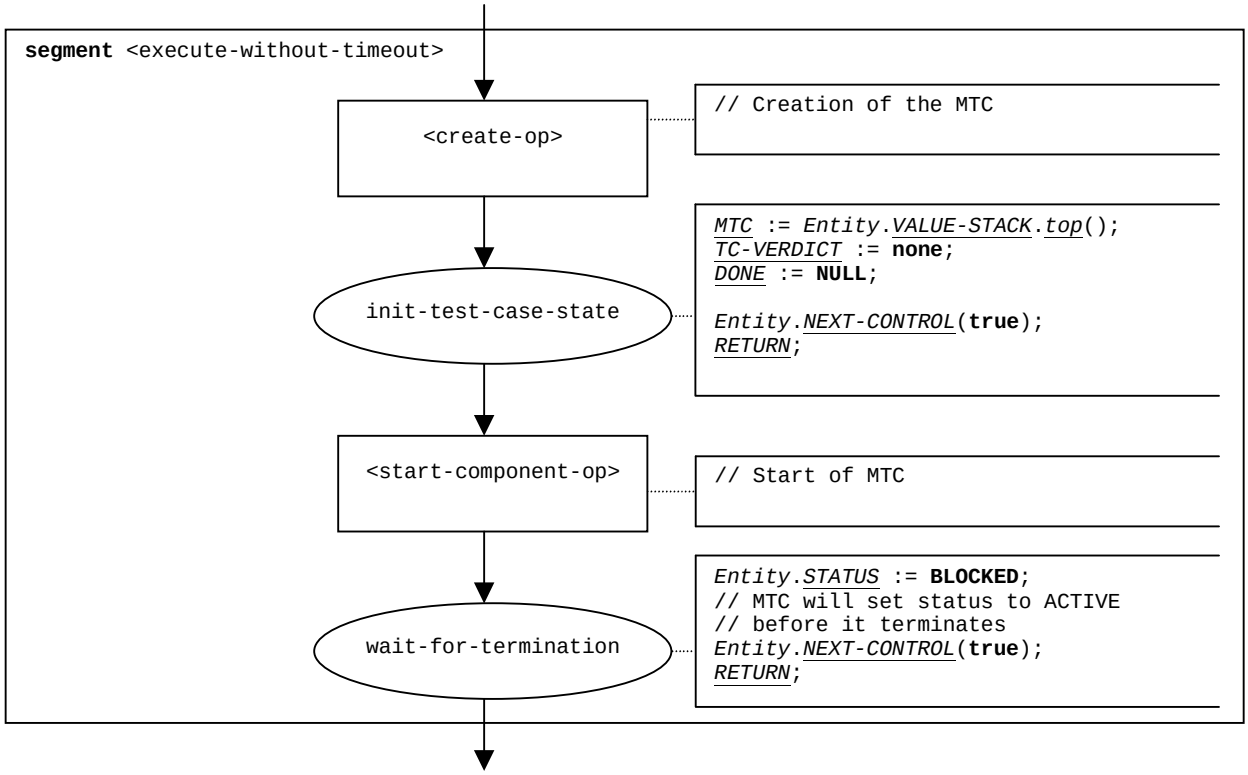

Figure 68: Flow graph segment <execute-without-timeout>

6.4 Replacements in Function call's operational semantics

6.4.0 Function call

The syntactical structure of a function call is:

```
<function-name>([<act-par-desc1>, … , <act-par-descn>]) |
apply(<function-expr>([<act-par-desc1>, … , <act-par-descn>])
```

The <function-name> and the <function-expr> denote to the identifier of the function that is invoked. The operational semantics assumes that <function-name> is a literal value of the function behaviour type. <act-par-descr$_1$>, … , <act-par-descr$_n$> describe the actual parameter values of the function call.

NOTE 1: A function call and an altstep call are handled in the same manner. Therefore, the altstep call (see clause 9.4 in ETSI ES 201 873-4 [2]) refers to this clause.

Furthermore, it is assumed that for each <act-par-desc$_1$> the corresponding formal parameter identifier <f-par-Id$_1$> is known, i.e. the syntactical structure above can be extended to:

```
<function-name>((<f-par-Id1>,<act-par-desc1>), … , (<f-par-Idn>,<act-par-descn>)) |
apply(<function-expr>((<f-par-Id1>,<act-par-desc1>), … , (<f-par-Idn>,<act-par-descn>)))
```

The flow graph segment <function-call> in figure 80 defines the execution of a function call. The execution is structured into three steps. In the first step, the identifier of the function to be called is determined. The identifier may directly be given by the function name or be given in form of an expression that evaluates to the function identifier. In the second step, a call record for the function is created. In the third step the values of the actual parameter are calculated and assigned to the corresponding fields in the call record. In the fourth step, the parameters called by reference are handled. In the fifth step, two situations have to be distinguished: the called function is a user-defined function (`<user-def-func-call>`), i.e. there exists a flow graph representation for the function, or the called function is a pre-defined or external function (`<predef-ext-func-call>`). In case of a user-defined function call, the control is given to the called function. In case of a pre-defined or external function, it is assumed that the call record can be used to execute the function in one step. The correct handling of reference parameters and return value (has to be pushed onto the value stack) is in the responsibility of the called function, i.e. is outside the scope of this operational semantics.

NOTE 2: If the function call models an altstep call, only the `<user-def-func-call>` branch will be chosen, because there exists a flow graph representation of the called altstep.

NOTE 3: The `<function call>` segment is also used to describe the start of the MTC in an **`execute`** statement. In this case, a call record for the test case is constructed and only the `<user-def-func-call>` branch will be chosen.

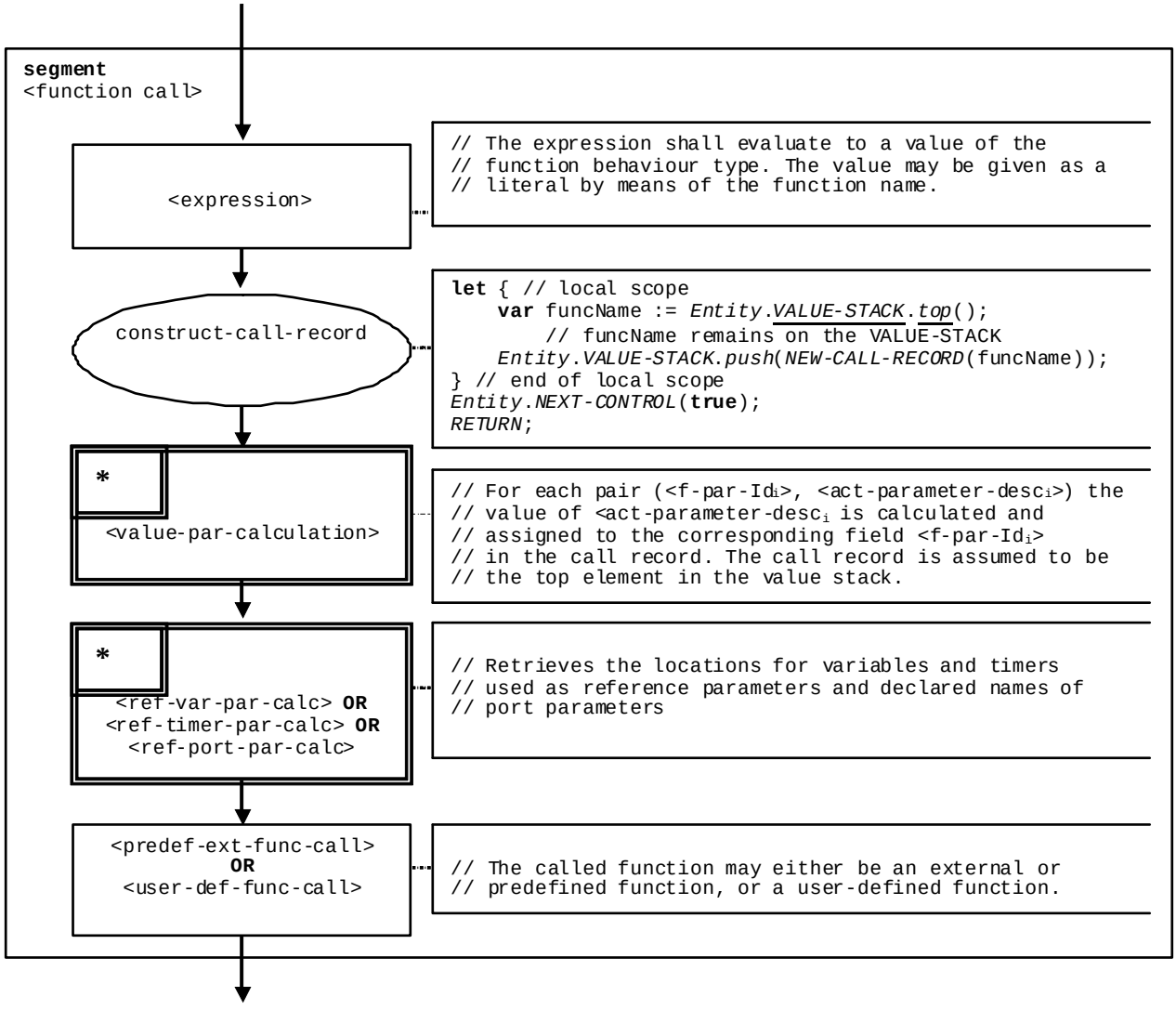

Figure 80: Flow graph segment <function-call>

6.4.1 Flow graph segment <user-def-func-call>

The flow graph-segment <user-def-func-call> (figure 84) describes the transfer of control to a called user-defined function.

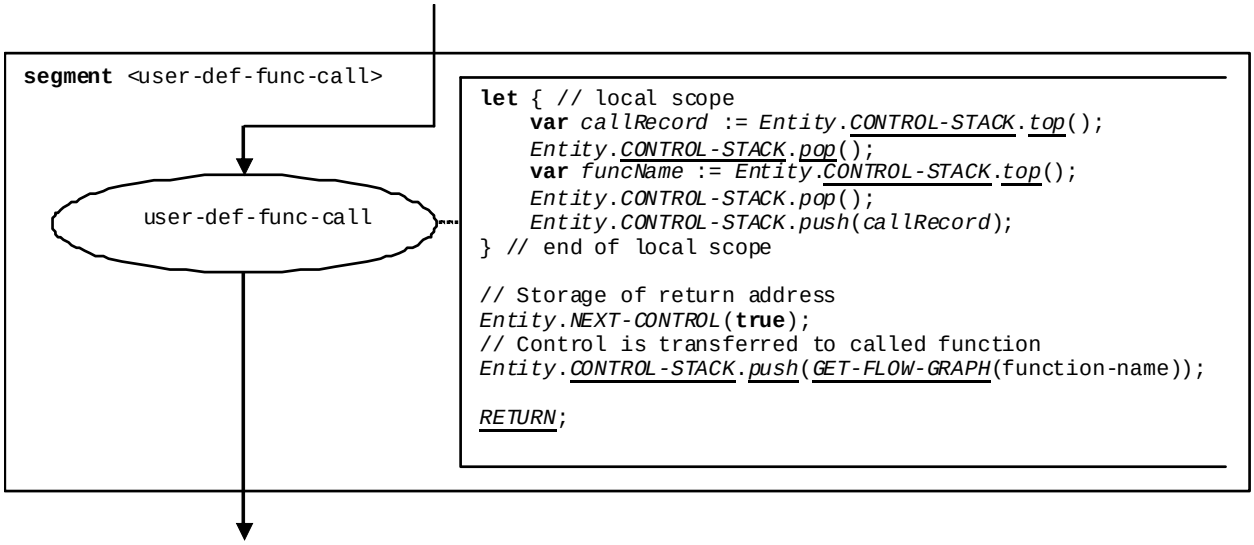

Figure 84: Flow graph segment <user-def-func-call>

6.4.2 Flow graph segment <predef-ext-func-call>

The flow graph-segment <predef-ext-func-call> (figure 85) describes the call of a pre-defined or external function.

NOTE: In figure 85, it is assumed that the call record can be used to execute the pre-defined or external function in one step.

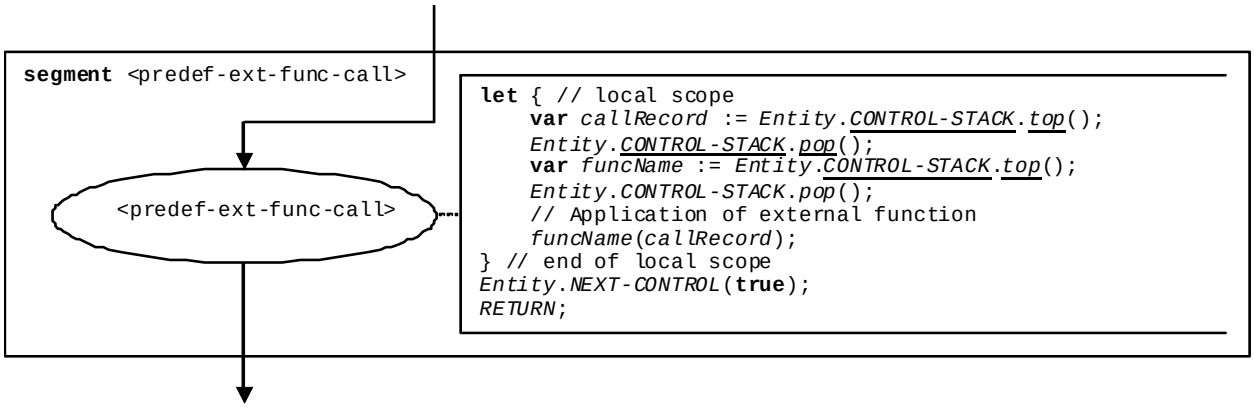

Figure 85: Flow graph segment <predef-ext-func-call>

6.5 Start component operation

The syntactical structure of the **start** component operation is:

```
<component-expression>.start(<function-name>(<act-par-desc1>,…, <act-par-descn>))    |
<component-expression>.start(apply(<function-expr>(<act-par-desc1>,…, <act-par-descn>))) |
<component-expression>.start(<deferred-function-behaviour>)
```

The **`start`** component operation starts a component. Using a component reference identifies the component to be started. The reference may be stored in a variable or be returned by a function, i.e. it is an expression that evaluates to a component reference.

The `<function-name>` and the `<function-expr>` denote the identifier of the function that defines the behaviour of the new component. <deferred-function-behaviour> denotes a reference to a deferred function behaviour. The operational semantics assumes that `<function-name>` is a literal name of a function and can be handled like the `<function-expr>`, i.e. as an expression that evaluates to a value of the function behaviour type. `<act-par-descr`$_1$`>`, …, `<act-par-descr`$_n$`>` provide the description of the actual parameter values of the `function`. The descriptions of the actual parameters are provided in form of expressions that have to be evaluated before the call can be executed. The handling of formal and actual value parameters is similar to their handling in function calls (see clause 9.24 in ETSI ES 201 873-4 [2]).

The flow graph segment <start-component-op> in figure 120 defines the execution of the **`start`** component operation. The start component operation is executed in five steps. In the first step, the identifier of the function to be started is determined. In the second step, a call record is created. In the third step the actual parameter values are calculated. In the fourth step the reference of the component to be started is retrieved, and, in the fifth step, control and call record are given to the new component.

NOTE: The flow graph segment in figure 120 includes the handling of reference parameters (`<ref-var-par-calc>`). Reference parameters are needed to explain reference parameters of test cases. The operational semantics assumes that these parameters are handled by the MTC.

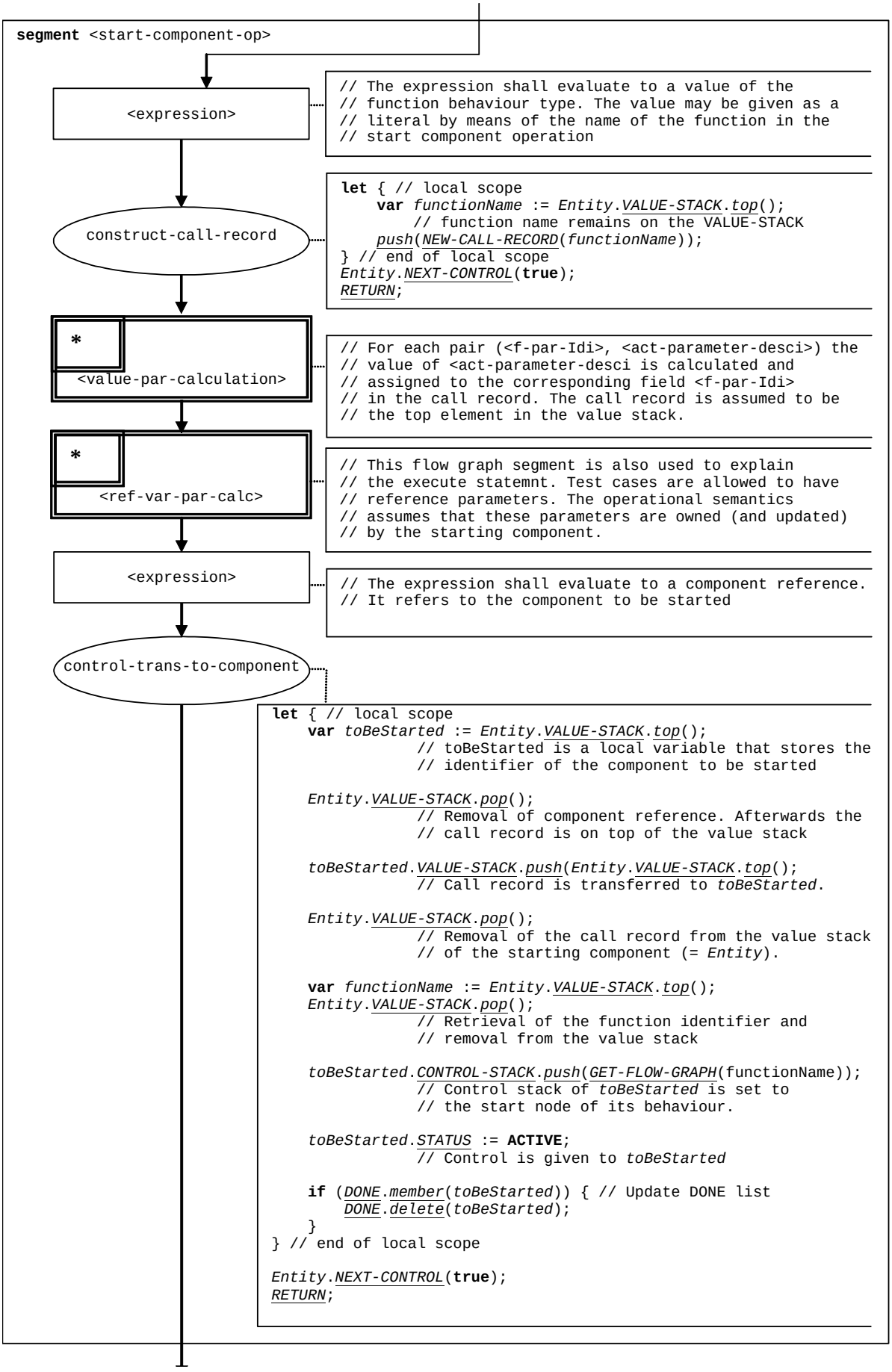

Figure 120: Flow graph segment <start-component-op>

7 TRI extensions for the package

This package does not have an effect on TRI.

8 TCI extensions for the package

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

Clause 7.2.2.1 Abstract TTCN-3 data types

Three additional type classes are used to distinguish the three kinds of behaviours: ALTSTEP, FUNCTION and TESTCASE They are also used for deferred altstep, function and testcase behaviour values, respectively.

`TciTypeClassType getTypeClass()` Returns the type class of the respective type. A value of `TciTypeClassType` can have one of the following constants: `ADDRESS, ALTSTEP, ANYTYPE, BITSTRING, BOOLEAN, CHARSTRING, COMPONENT, ENUMERATED, FLOAT, FUNCTION, HEXSTRING, INTEGER, OBJID, OCTETSTRING, RECORD, RECORD_OF, SET, SET_OF, TESTCASE, UNION, UNIVERSAL_CHARSTRING, VERDICT.`

Clause 7.2.2.2 Abstract TTCN-3 values

Clause 7.2.2.2 Abstract TTCN-3 values is extended by figure 4.

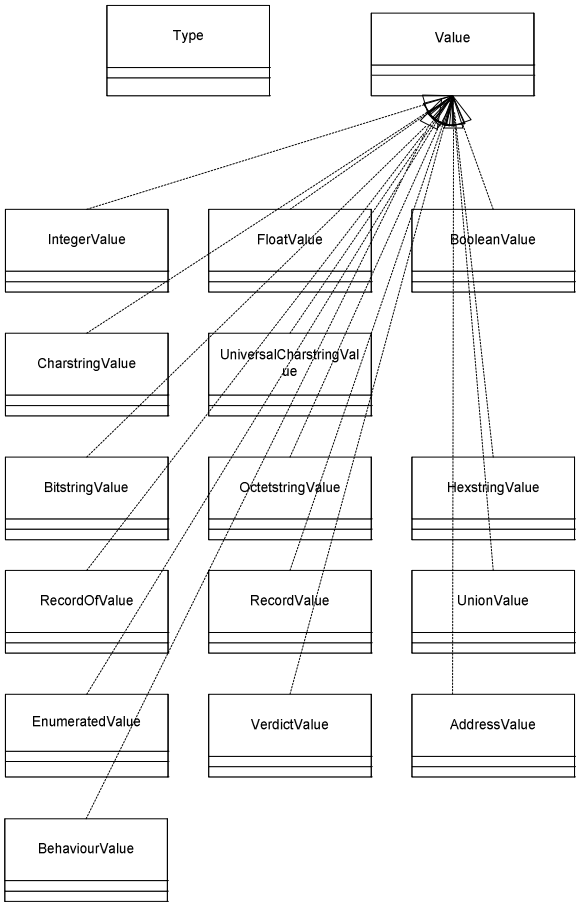

Figure 4: Hierarchy of abstract values

7.2.2.2.16 The abstract data type BehaviourValue

The following operations are defined on the base abstract data type `BehaviourValue`, which is used to represent both behaviour values and deferred behaviour values. The concrete representations of these operations are defined in the respective language mapping sections:

`Value getName()`	Returns the name of the behaviour.
`TciModuleIdType getDefiningModule()`	Returns the module identifier of the module in which the behaviour is defined. Returns the distinct value `null` if the behaviour is a predefined function.

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

Clause 8.3.2.4 TciTypeClassType

Clause 8.3.2.4 TciTypeClassType shall be extended as follows:

`TciTypeClassType` is mapped to the following interface:

```
// TCI IDL TciTypeClassType
package org.etsi.ttcn.tci;
public interface TciTypeClass {
    public final static int ADDRESS             = 0 ;
    public final static int ANYTYPE             = 1 ;
    public final static int BITSTRING               = 2 ;
    public final static int BOOLEAN             = 3 ;
    public final static int CHARSTRING              = 5 ;
    public final static int COMPONENT               = 6 ;
    public final static int ENUMERATED              = 7 ;
    public final static int FLOAT                   = 8 ;
    public final static int HEXSTRING               = 9 ;
    public final static int INTEGER             = 10 ;
    public final static int OBJID                   = 11 ;
    public final static int OCTETSTRING         = 12 ;
    public final static int RECORD                  = 13 ;
    public final static int RECORD_OF               = 14 ;
    public final static int SET                 = 15 ;
    public final static int SET_OF                  = 16 ;
    public final static int UNION                   = 17 ;
    public final static int UNIVERSAL_CHARSTRING    = 19 ;
    public final static int VERDICT             = 20 ;
    public final static int ALTSTEP             = 21 ;
    public final static int FUNCTION                = 22 ;
    public final static int TESTCASE                = 23 ;
}
```

Clause 8.3.4 Abstract value mapping

Clause 8.3.4 Abstract value mapping shall be extended by:

8.3.4.16 BehaviourValue

`BehaviourValue` is mapped to the following interface:

```
// TCI IDL Type
package org.etsi.ttcn.tci;
public interface BehaviourValue {
    public String       getName ();
    public TciModuleId  getDefiningModule ();
}
```

Methods:

- `getName()` Returns the name of the behaviour as defined in the TTCN-3 module.
- `getDefiningModule()` Returns the module identifier of the module the behaviour has been defined in.

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

Clause 9.2 Value Interfaces

Clause 9.2 Value Interfaces shall be extended by:

TCI IDL Interface	ANSI C representation	Notes and comments
BehaviourValue		
Value getName()	`Value tciName(Value inst)`	Returns the name of the behaviour as a CharstringValue.
TciModuleIdType getDefiningModule()	`TciModuleIdType tciDefiningModule (Value inst)`	Returns the module identifier of the module in which the behaviour is defined. The module identifier will be empty in case the behaviour is a predefined function.

Clause 9.5 Data

Clause 9.5 Data shall be extended by:

TCI IDL ADT	ANSI C representation (Type definition)	Notes and comments

TciTypeClassType	`typedef enum`
	`{`
	`  TCI_ADDRESS_TYPE,`
	`  TCI_ANYTYPE_TYPE,`
	`  TCI_BITSTRING_TYPE,`
	`  TCI_BOOLEAN_TYPE,`
	`  TCI_CHAR_TYPE,`
	`  TCI_CHARSTRING_TYPE,`
	`  TCI_COMPONENT_TYPE,`
	`  TCI_ENUMERATED_TYPE,`
	`  TCI_FLOAT_TYPE,`
	`  TCI_HEXSTRING_TYPE,`
	`  TCI_INTEGER_TYPE,`
	`  TCI_OBJID_TYPE,`
	`  TCI_OCTETSTRING_TYPE,`
	`  TCI_RECORD_TYPE,`
	`  TCI_RECORD_OF_TYPE,`
	`  TCI_SET_TYPE,`
	`  TCI_SET_OF_TYPE,`
	`  TCI_UNION_TYPE,`
	`  TCI_UNIVERSAL_CHAR_TYPE,`
	`  TCI_UNIVERSAL_CHARSTRING_TYPE,`
	`  TCI_VERDICT_TYPE,`
	`  TCI_ALTSTEP_TYPE,`
	`  TCI_FUNCTION_TYPE,`
	`  TCI_TESTCASE_TYPE`
	`} TciTypeClassType;`
		TCI_ALTSTEP_TYPE, TCI_FUNCTION_TYPE, and TCI_TESTCASE_TYPE added.

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

Clause 10.5.2.14 TciTypeClass

Clause 10.5.2.14 TciTypeClass shall be extended by the three behaviour type classes as follows:

Defines the different type classes in TTCN-3 (i.e. boolean, float, etc.). It is mapped to the following pure virtual class:

```
class TciTypeClass {
public:
    static const TciTypeClass TCI_ADDRESS
    static const TciTypeClass TCI_ANYTYPE
    static const TciTypeClass TCI_BITSTRING
```

```
    static const TciTypeClass TCI_BOOLEAN
    static const TciTypeClass TCI_CHARSTRING
    static const TciTypeClass TCI_COMPONENT
    static const TciTypeClass TCI_ENUMERATED
    static const TciTypeClass TCI_FLOAT
    static const TciTypeClass TCI_HEXSTRING
    static const TciTypeClass TCI_INTEGER
    static const TciTypeClass TCI_OCTETSTRING
    static const TciTypeClass TCI_RECORD
    static const TciTypeClass TCI_RECORD_OF
    static const TciTypeClass TCI_ARRAY
    static const TciTypeClass TCI_SET
    static const TciTypeClass TCI_SET_OF
    static const TciTypeClass TCI_UNION
    static const TciTypeClass TCI_UNIVERSAL_CHARSTRING
    static const TciTypeClass TCI_VERDICT
    static const TciTypeClass TCI_ALTSTEP
    static const TciTypeClass TCI_FUNCTION
    static const TciTypeClass TCI_TESTCASE
    ~TciTypeClass ()
    Tboolean equals (const TciTypeClass &p_other) const
    Tstring toString () const
    Tinteger getTypeClass() const
}
```

Clause 10.5.3 Abstract TTCN-3 data types and values

Clause 10.5.3 Abstract TTCN-3 data types and values shall be extended by:

10.5.3.18 BehaviourValue

A value of BehaviourValue represents an altstep, function or testcase in a TTCN-3 module. It is mapped to the following pure virtual class:

```
class BehaviourValue {
public:
    virtual ~BehaviourValue ();
    virtual const TciModuleId & getDefiningModule () const =0;
    virtual const Tstring & getName () const =0;
    virtual Tboolean equals (const BehaviourValue &behVal) const =0;
    virtual BehaviourValue * cloneBehaviourValue () const =0;
    virtual Tboolean operator< (const BehaviourValue &behVal) const =0;
}
```

10.5.3.18.1 Methods

- `~TciType ()`

  Destructor.

- `getDefiningModule ()`

  Return the defining module as defined in the TTCN-3 ATS.

- `getName ()`

  Return behaviour name as defined in the ATS.

- `equals (const BehaviourValue &behVal)`

  Return true if the behaviours are equal.

- `cloneType ()`

  Return a copy of the BehaviourValue.

- `operator< (const BehaviourValue &behVal)`

  Operator < overload.

8.5 Extensions to ETSI ES 201 873-6, clause 11 (W3C XML mapping)

Clause 11.3.3.1 Value

Clause 11.3.3.1 Value shall be extended by:

`Value` is mapped to the following complex type:

```
<xsd:complexType name="Value" mixed="true">
    <xsd:choice>
        <xsd:element name="integer" type="Values:IntegerValue"/>
        <xsd:element name="float" type="Values:FloatValue"/>
        <xsd:element name="boolean" type="Values:BooleanValue"/>
        <xsd:element name="verdicttype" type="Values:VerdictValue"/>
        <xsd:element name="bitstring" type="Values:BitstringValue"/>
        <xsd:element name="hexstring" type="Values:HexstringValue"/>
        <xsd:element name="octetstring" type="Values:OctetstringValue"/>
        <xsd:element name="charstring" type="Values:CharstringValue"/>
        <xsd:element name="universal_charstring" type="Values:UniversalCharstringValue"/>
        <xsd:element name="record" type="Values:RecordValue"/>
        <xsd:element name="record_of" type="Values:RecordOfValue"/>
        <xsd:element name="array" type="Values:ArrayValue"/>
        <xsd:element name="set" type="Values:SetValue"/>
        <xsd:element name="set_of" type="Values:SetOfValue"/>
        <xsd:element name="enumerated" type="Values:EnumeratedValue"/>
        <xsd:element name="union" type="Values:UnionValue"/>
        <xsd:element name="anytype" type="Values:AnytypeValue"/>
        <xsd:element name="address" type="Values:AddressValue"/>
        <xsd:element name="behaviour" type="Values:BehaviourValue"/>
    </xsd:choice>
    <xsd:attributeGroup ref="Values:ValueAtts"/>
</xsd:complexType>

<xsd:attributeGroup name="ValueAtts">
    <xsd:attribute name="name" type="SimpleTypes:TString" use="optional"/>
    <xsd:attribute name="type" type="SimpleTypes:TString" use="optional"/>
    <xsd:attribute name="module" type="SimpleTypes:TString" use="optional"/>
    <xsd:attribute name="annotation" type="SimpleTypes:TString" use="optional"/>
</xsd:attributeGroup>
```

Choice of Elements:

- `integer` An integer value.
- `float` A float value.
- `boolean` A boolean value.
- `verdicttype` A verdicttype value.
- `bitstring` A bitstring value.
- `hexstring` An hexstring value.
- `octetstring` An octetstring value.
- `charstring` A charstring value.
- `universal_charstring` A universal charstring value.
- `record` A record value.
- `record_of` A record of value.
- `array` An array value.
- `set` A set value.

- `set_of` A set of value.
- `enumerated` An enumerated value.
- `union` A union value.
- `anytype` An anytype value.
- `address` An address value.
- `behaviour` A behaviour value.

Attributes:

- `name` The name of the value, if known.
- `type` The type of the value, if known.
- `module` The module of the value, if known.
- `annotation` A helper attribute to provide additional matching/mismatching information, etc.

11.3.3.24b BehaviourValue

`BehaviourValue` is mapped to the following complex type:

```
<xsd:complexType name="BehaviourValue">
    <xsd:sequence>
        <xsd:element name="name" type="Types:QualifiedName"/>
    </xsd:sequence>
</xsd:complexType>
```

Elements:

- `name` The qualified name of the behaviour.

Attributes:

- none.

Clause 11.3.3.12 RecordValue

All clauses for structured values are to be extended with an element for a BehaviourValue.

11.3.3.12 RecordValue

`RecordValue` is mapped to the following complex type:

```
<xsd:complexType name="RecordValue">
    <xsd:choice>
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="integer" type="Values:IntegerValue"/>
            <xsd:element name="float" type="Values:FloatValue"/>
            <xsd:element name="boolean" type="Values:BooleanValue"/>
            <xsd:element name="verdicttype" type="Values:VerdictValue"/>
            <xsd:element name="bitstring" type="Values:BitstringValue"/>
            <xsd:element name="hexstring" type="Values:HexstringValue"/>
            <xsd:element name="octetstring" type="Values:OctetstringValue"/>
            <xsd:element name="charstring" type="Values:CharstringValue"/>
            <xsd:element name="universal_charstring"
                type="Values:UniversalCharstringValue"/>
            <xsd:element name="record" type="Values:RecordValue"/>
            <xsd:element name="record_of" type="Values:RecordOfValue"/>
            <xsd:element name="array" type="Values:ArrayValue"/>
            <xsd:element name="set" type="Values:SetValue"/>
            <xsd:element name="set_of" type="Values:SetOfValue"/>
            <xsd:element name="enumerated" type="Values:EnumeratedValue"/>
            <xsd:element name="union" type="Values:UnionValue"/>
            <xsd:element name="anytype" type="Values:AnytypeValue"/>
```

```
            <xsd:element name="address" type="Values:AddressValue"/>
            <xsd:element name="behaviour" type="Values:BehaviourValue"/>
        </xsd:choice>
        <xsd:element name="null" type="Templates:null"/>
        <xsd:element name="omit" type="Templates:omit"/>
    </xsd: choice>
    <xsd:attributeGroup ref="Values:ValueAtts"/>
</xsd:complexType>
```

Sequence of Elements:

- `integer` An integer value.
- `float` A float value.
- `boolean` A boolean value.
- `verdicttype` A verdicttype value.
- `bitstring` A bitstring value.
- `hexstring` An hexstring value.
- `octetstring` An octetstring value.
- `charstring` A charstring value.
- `universal_charstring` A universal charstring value.
- `record` A record value.
- `record_of` A record of value.
- `array` An array value.
- `set` A set value.
- `set_of` A set of value.
- `enumerated` An enumerated value.
- `union` A union value.
- `anytype` An anytype value.
- `address` An address value.
- `behaviour` A behaviour value.
- `null` If no field is given.
- `omit` If the field is omitted.

Attributes:

- The same attributes as those of Value.

11.3.3.13 RecordOfValue

`RecordOfValue` is mapped to the following complex type:

```
<xsd:complexType name="RecordOfValue">
    <xsd:choice>
        <xsd:element name="integer" type="Values:IntegerValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="float" type="Values:FloatValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="boolean" type="Values:BooleanValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="verdicttype" type="Values:VerdictValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="bitstring" type="Values:BitstringValue"
                minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="hexstring" type="Values:HexstringValue"
                minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="octetstring" type="Values:OctetstringValue"
                minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="charstring" type="Values:CharstringValue"
                minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="universal_charstring"
                type="Values:UniversalCharstringValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="record" type="Values:RecordValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="record_of" type="Values:RecordOfValue"
                minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="array" type="Values:ArrayValue"
                minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="set" type="Values:SetValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="set_of" type="Values:SetOfValue"
                minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="enumerated" type="Values:EnumeratedValue"
                minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="union" type="Values:UnionValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="anytype" type="Values:AnytypeValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="address" type="Values:AddressValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="behaviour" type="Values:BehaviourValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="null" type="Templates:null"/>
        <xsd:element name="omit" type="Templates:omit"/>
    </xsd:choice>
    <xsd:attributeGroup ref="Values:ValueAtts"/>
</xsd:complexType>
```

Choice of Sequence of Elements:

- `integer` An integer value.
- `float` A float value.
- `boolean` A boolean value.
- `verdicttype` A verdicttype value.
- `bitstring` A bitstring value.
- `hexstring` An hexstring value.
- `octetstring` An octetstring value.
- `charstring` A charstring value.
- `universal_charstring` A universal charstring value.
- `record` A record value.

- `record_of` A record of value.
- `array` An array value.
- `set` A set value.
- `set_of` A set of value.
- `enumerated` An enumerated value.
- `union` A union value.
- `anytype` An anytype value.
- `address` An address value.
- `behaviour` A behaviour value.
- `null` If no field is given.
- `omit` If the field is omitted.

Attributes:

- The same attributes as those of Value.

11.3.3.14 ArrayValue

`ArrayValue` is mapped to the following complex type:

```
<xsd:complexType name="ArrayValue">
    <xsd:choice>
        <xsd:element name="integer" type="Values:IntegerValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="float" type="Values:FloatValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="boolean" type="Values:BooleanValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="verdicttype" type="Values:VerdictValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="bitstring" type="Values:BitstringValue"
                minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="hexstring" type="Values:HexstringValue"
                minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="octetstring" type="Values:OctetstringValue"
                minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="charstring" type="Values:CharstringValue"
                minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="universal_charstring"
                type="Values:UniversalCharstringValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="record" type="Values:RecordValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="record_of" type="Values:RecordOfValue"
                minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="array" type="Values:ArrayValue"
                minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="set" type="Values:SetValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="set_of" type="Values:SetOfValue"
                minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="enumerated" type="Values:EnumeratedValue"
                minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="union" type="Values:UnionValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="anytype" type="Values:AnytypeValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="address" type="Values:AddressValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="behaviour" type="Values:BehaviourValue" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element name="null" type="Templates:null"/>
```

```
        <xsd:element name="omit" type="Templates:omit"/>
    </xsd:choice>
    <xsd:attributeGroup ref="Values:ValueAtts"/>
</xsd:complexType>
```

Choice of Sequence of Elements:

- `integer` An integer value.
- `float` A float value.
- `boolean` A boolean value.
- `verdicttype` A verdicttype value.
- `bitstring` A bitstring value.
- `hexstring` An hexstring value.
- `octetstring` An octetstring value.
- `charstring` A charstring value.
- `universal_charstring` A universal charstring value.
- `record` A record value.
- `record_of` A record of value.
- `array` An array value.
- `set` A set value.
- `set_of` A set of value.
- `enumerated` An enumerated value.
- `union` A union value.
- `anytype` An anytype value.
- `address` An address value.
- `behaviour` A behaviour value.
- `null` If no field is given.
- `omit` If the field is omitted.

Attributes:

- The same attributes as those of Value.

11.3.3.15 SetValue

`SetValue` is mapped to the following complex type:

```
<xsd:complexType name="SetValue">
    <xsd:choice>
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="integer" type="Values:IntegerValue"/>
            <xsd:element name="float" type="Values:FloatValue"/>
            <xsd:element name="boolean" type="Values:BooleanValue"/>
            <xsd:element name="verdicttype" type="Values:VerdictValue"/>
            <xsd:element name="bitstring" type="Values:BitstringValue"/>
            <xsd:element name="hexstring" type="Values:HexstringValue"/>
            <xsd:element name="octetstring" type="Values:OctetstringValue"/>
            <xsd:element name="charstring" type="Values:CharstringValue"/>
```

```
            <xsd:element name="universal_charstring"
                type="Values:UniversalCharstringValue"/>
            <xsd:element name="record" type="Values:RecordValue"/>
            <xsd:element name="record_of" type="Values:RecordOfValue"/>
            <xsd:element name="array" type="Values:ArrayValue"/>
            <xsd:element name="set" type="Values:SetValue"/>
            <xsd:element name="set_of" type="Values:SetOfValue"/>
            <xsd:element name="enumerated" type="Values:EnumeratedValue"/>
            <xsd:element name="union" type="Values:UnionValue"/>
            <xsd:element name="anytype" type="Values:AnytypeValue"/>
            <xsd:element name="address" type="Values:AddressValue"/>
            <xsd:element name="behaviour" type="Values:BehaviourValue"/>
        </xsd:choice>
        <xsd:element name="null" type="Templates:null"/>
        <xsd:element name="omit" type="Templates:omit"/>
    </xsd:choice>
    <xsd:attributeGroup ref="Values:ValueAtts"/>
</xsd:complexType>
```

Sequence of Elements:

- `integer` An integer value.
- `float` A float value.
- `boolean` A boolean value.
- `verdicttype` A verdicttype value.
- `bitstring` A bitstring value.
- `hexstring` An hexstring value.
- `octetstring` An octetstring value.
- `charstring` A charstring value.
- `universal_charstring` A universal charstring value.
- `record` A record value.
- `record_of` A record of value.
- `array` An array value.
- `set` A set value.
- `set_of` A set of value.
- `enumerated` An enumerated value.
- `union` A union value.
- `anytype` An anytype value.
- `address` An address value.
- `behaviour` A behaviour value.
- `null` If no field is given.
- `omit` If the field is omitted.

Attributes:

- The same attributes as those of Value.

11.3.3.16 SetOfValue

`SetOfValue` is mapped to the following complex type:

```
<xsd:complexType name="SetOfValue">
    <xsd:choice>
        <xsd:element name="integer" type="Values:IntegerValue" minOccurs="0"
            maxOccurs="unbounded"/>
        <xsd:element name="float" type="Values:FloatValue" minOccurs="0"
            maxOccurs="unbounded"/>
        <xsd:element name="boolean" type="Values:BooleanValue" minOccurs="0"
            maxOccurs="unbounded"/>
        <xsd:element name="verdicttype" type="Values:VerdictValue" minOccurs="0"
            maxOccurs="unbounded"/>
        <xsd:element name="bitstring" type="Values:BitstringValue"
            minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="hexstring" type="Values:HexstringValue"
            minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="octetstring" type="Values:OctetstringValue"
            minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="charstring" type="Values:CharstringValue"
            minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="universal_charstring"
            type="Values:UniversalCharstringValue" minOccurs="0"
            maxOccurs="unbounded"/>
        <xsd:element name="record" type="Values:RecordValue" minOccurs="0"
            maxOccurs="unbounded"/>
        <xsd:element name="record_of" type="Values:RecordOfValue"
            minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="array" type="Values:ArrayValue"
            minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="set" type="Values:SetValue" minOccurs="0"
            maxOccurs="unbounded"/>
        <xsd:element name="set_of" type="Values:SetOfValue"
            minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="enumerated" type="Values:EnumeratedValue"
            minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="union" type="Values:UnionValue" minOccurs="0"
            maxOccurs="unbounded"/>
        <xsd:element name="anytype" type="Values:AnytypeValue" minOccurs="0"
            maxOccurs="unbounded"/>
        <xsd:element name="address" type="Values:AddressValue" minOccurs="0"
            maxOccurs="unbounded"/>
        <xsd:element name="behaviour" type="Values:BehaviourValue" minOccurs="0"
            maxOccurs="unbounded"/>
        <xsd:element name="null" type="Templates:null"/>
        <xsd:element name="omit" type="Templates:omit"/>        </xsd:choice>
    <xsd:attributeGroup ref="Values:ValueAtts"/>
</xsd:complexType>
```

Choice of Sequence of Elements:

- `integer` An integer value.
- `float` A float value.
- `boolean` A boolean value.
- `verdicttype` A verdicttype value.
- `bitstring` A bitstring value.
- `hexstring` An hexstring value.
- `octetstring` An octetstring value.
- `charstring` A charstring value.
- `universal_charstring` A universal charstring value.
- `record` A record value.
- `record_of` A record of value.

- `array` An array value.
- `set` A set value.
- `set_of` A set of value.
- `enumerated` An enumerated value.
- `union` A union value.
- `anytype` An anytype value.
- `address` An address value.
- `behaviour` A behaviour value.
- `null` If no field is given.
- `omit` If the field is omitted.

Attributes:

- The same attributes as those of Value.

11.3.3.18 UnionValue

`UnionValue` is mapped to the following complex type:

```
<xsd:complexType name="UnionValue">
    <xsd:choice>
        <xsd:element name="integer" type="Values:IntegerValue"/>
        <xsd:element name="float" type="Values:FloatValue"/>
        <xsd:element name="boolean" type="Values:BooleanValue"/>
        <xsd:element name="verdicttype" type="Values:VerdictValue"/>
        <xsd:element name="bitstring" type="Values:BitstringValue"/>
        <xsd:element name="hexstring" type="Values:HexstringValue"/>
        <xsd:element name="octetstring" type="Values:OctetstringValue"/>
        <xsd:element name="charstring" type="Values:CharstringValue"/>
        <xsd:element name="universal_charstring"
            type="Values:UniversalCharstringValue"/>
        <xsd:element name="record" type="Values:RecordValue"/>
        <xsd:element name="record_of" type="Values:RecordOfValue"/>
        <xsd:element name="array" type="Values:ArrayValue"/>
        <xsd:element name="set" type="Values:SetValue"/>
        <xsd:element name="set_of" type="Values:SetOfValue"/>
        <xsd:element name="enumerated" type="Values:EnumeratedValue"/>
        <xsd:element name="union" type="Values:UnionValue"/>
        <xsd:element name="anytype" type="Values:AnytypeValue"/>
        <xsd:element name="address" type="Values:AddressValue"/>
        <xsd:element name="behaviour" type="Values:BehaviourValue"/>
        <xsd:element name="null" type="Templates:null"/>
        <xsd:element name="omit" type="Templates:omit"/>
    </xsd:choice>
    <xsd:attributeGroup ref="Values:ValueAtts"/>
</xsd:complexType>
```

Choice of Elements:

- `integer` An integer value.
- `float` A float value.
- `boolean` A boolean value.
- `verdicttype` A verdicttype value.
- `bitstring` A bitstring value.
- `hexstring` An hexstring value.

- `octetstring` An octetstring value.
- `charstring` A charstring value.
- `universal_charstring` A universal charstring value.
- `record` A record value.
- `record_of` A record of value.
- `array` An array value.
- `set` A set value.
- `set_of` A set of value.
- `enumerated` An enumerated value.
- `union` A union value.
- `anytype` An anytype value.
- `address` An address value.
- `behaviour` A behaviour value.

Attributes:

- The same attributes as those of Value.

11.3.3.19 AnytypeValue

`AnytypeValue` is mapped to the following complex type:

```
<xsd:complexType name="AnytypeValue">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="integer" type="Values:IntegerValue"/>
        <xsd:element name="float" type="Values:FloatValue"/>
        <xsd:element name="boolean" type="Values:BooleanValue"/>
        <xsd:element name="verdicttype" type="Values:VerdictValue"/>
        <xsd:element name="bitstring" type="Values:BitstringValue"/>
        <xsd:element name="hexstring" type="Values:HexstringValue"/>
        <xsd:element name="octetstring" type="Values:OctetstringValue"/>
        <xsd:element name="charstring" type="Values:OctetstringValue"/>
        <xsd:element name="universal_charstring"
            type="Values:UniversalCharstringValue"/>
        <xsd:element name="record" type="Values:RecordValue"/>
        <xsd:element name="record_of" type="Values:RecordOfValue"/>
        <xsd:element name="array" type="Values:ArrayValue"/>
        <xsd:element name="set" type="Values:SetValue"/>
        <xsd:element name="set_of" type="Values:SetOfValue"/>
        <xsd:element name="enumerated" type="Values:EnumeratedValue"/>
        <xsd:element name="union" type="Values:UnionValue"/>
        <xsd:element name="address" type="Values:AddressValue"/>
        <xsd:element name="behaviour" type="Values:BehaviourValue"/>
        <xsd:element name="null" type="Templates:null"/>
        <xsd:element name="omit" type="Templates:omit"/>
    </xsd:choice>
    <xsd:attributeGroup ref="Values:ValueAtts"/>
</xsd:complexType>
```

Choice of Elements:

- `integer` An integer value.
- `float` A float value.
- `boolean` A boolean value.
- `verdicttype` A verdicttype value.

- `bitstring` A bitstring value.
- `hexstring` An hexstring value.
- `octetstring` An octetstring value.
- `charstring` A charstring value.
- `universal_charstring` A universal charstring value.
- `record` A record value.
- `record_of` A record of value.
- `array` An array value.
- `set` A set value.
- `set_of` A set of value.
- `enumerated` An enumerated value.
- `union` A union value.
- `address` An address value.
- `behaviour` A behaviour value.

Attributes:

- The same attributes as those of Value.

9 Extensions to TTCN-3 documentation comment specification for the package

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

Add ", function behaviour types" to the cell "Functions (TTCN-3 and external)", ", altstep behaviour types" to the cell "Altsteps" and ", test case behaviour types" to the cell "Test cases" in the header of table A.1 of [7].

Table A.1 shows an example for the extended table A.1. Note that other packages may also extend the same table. Underlined character shows the insertions by this package.

Table A.1: Relation of documentation tags and TTCN-3

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

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

History

Document history		
V1.1.1	January 2010	Publication
V1.2.1	May 2011	Publication
V1.3.1	April 2013	Publication
V1.4.1	June 2015	Publication
V1.5.1	August 2017	Publication
V1.6.1	May 2018	Publication
V1.7.1	May 2020	Publication
V1.8.1	June 2021	Publication
V1.9.1	February 2022	Membership Approval Procedure MV 20220429: 2022-02-28 to 2022-04-29
V1.9.1	May 2022	Publication