Final draft ETSI ES 203 022 V1.3.1 (2019-02)

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

Reference
RES/MTS-203022-AdvMatchv131

Keywords
conformance, testing, TTCN-3

ETSI

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

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

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

Contents

Intellectual Property Rights

Essential patents

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

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

Trademarks

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

Foreword

This final draft ETSI Standard (ES) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS), and is now submitted for the ETSI standards Membership Approval Procedure.

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

Part 1: "TTCN-3 Core Language";

Part 4: "TTCN-3 Operational Semantics";

Part 5: "TTCN-3 Runtime Interface (TRI)";

Part 6: "TTCN-3 Control Interface (TCI)";

Part 7: "Using ASN.1 with TTCN-3";

Part 8: "The IDL to TTCN-3 Mapping";

Part 9: "Using XML schema with TTCN-3";

Part 10: "TTCN-3 Documentation Comment Specification";

Part 11: "Using JSON with TTCN-3".

NOTE: Part 2 is in status "historical" and part 3 is no longer maintained.

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 of advance matching 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 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, 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] and ETSI ES 201 873-6 [4] apply.

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] 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-3:2017 Advanced Matching"` - 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 4.9.1;
- 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 advanced matching mechanisms for TTCN-3, i.e. new matching mechanisms which go beyond the matching mechanisms defined in ETSI ES 201 873-1 [1]. This package realizes the following concepts:

- Dynamic matching: allows to specify matching in a function-like fashion, i.e. statement blocks and functions can be used to define matching.
- Templates with variable bindings: ease the retrieval field values of received messages and signatures. For this the "->" symbol is used to denote a value assignment to a variable or an **out** parameter in the scope of a template definition in case of a successful template matching.
- Additional logical operators: conjunction, implication, exclusion and disjunction allow the combination matching mechanisms for advanced matching.
- Repetition: allows to match repetitions of a sub-sequence templates inside values of a certain type.
- Restrictions for the omit symbol and templates with restriction omit are relieved allowing omit symbols and templates with restriction omit as operands for the equality operator.

5.1 Dynamic Matching

A dynamic matching is a special matching mechanism. Similar to other matching mechanisms, it can be considered as a Boolean function that indicates successful matching for the value to be matched by returning the value **`true`** and unsuccessful matching by returning the value **`false`**.

Syntactical Structure

@dynamic (*StatementBlock* | *FunctionRef*)

Semantic Description

The *StatementBlock* shall return a value of type **`boolean`**. The value to be matched is referenced by the special keyword **`value`**. When applying this matching mechanism to a value, the *StatementBlock* is executed and if and only if the execution returns **`true`**, the dynamic matching function matches. Unsuccessful matching shall return **`false`**.

A dynamic matching function can only be used in the context of a template, the **`value`** expression inside the *StatementBlock* shall have the same type as the whole template.

The notation **`@dynamic`** *FunctionRef* denotes a shorthand for the special case **`@dynamic { return`** *FunctionRef***`(value) }`** where *FunctionRef* is a reference to a Boolean function with the first parameter compatible with the template's type. If the function contains more than one parameter, all parameters following the first one shall have a default value, The type of the first parameter of the referenced function determines the type context, if this template's place of usage does not provide a type context.

Restrictions

a) The dynamic matching syntax shall only be used in a typed context.

b) The *StatementBlock* shall compute a value of type **`boolean`**.

c) The *StatementBlock* shall be deterministic and side-effect free and follow the restrictions of clause 16.1.4 of ETSI ES 201 873-1 [1].

d) The *StatementBlock* shall not use variables that are declared outside of the *StatementBlock*.

e) The *StatementBlock* shall not use **`inout`** or **`out`** parameters.

f) Only if the dynamic matching syntax appears on the right-hand-side of a parameterized template definition, the formal **in** parameters of that definition may be referenced inside the *StatementBlock*. All other formal **in** parameters shall not be used by the *StatementBlock*.

g) The *FunctionRef* shall not have a **runs on** clause and the *StatementBlock* shall not use any behaviour that has a **runs on** clause.

EXAMPLE:

```
    type record of integer Numbers;
    template Numbers mw_sorted := @dynamic { // value is of type Numbers
        for (var integer v_i := 1; v_i < lengthof(value); v_i := v_i + 1)  {
            if (value[v_i-1] > value[v_i]) { return false }
        }
        return true;
    }   // mw_sorted(v_recInt) matches all values of type Numbers
// if elements of v_recInt do not break an ascending order

    type record Coordinate { float x, float y };
    external function @deterministic fx_distance(Coordinate p_a, Coordinate p_b) return float;
    template float mw_closeTo(Coordinate p_origin := { 0.0, 0.0 }, float p_maxDistance := 1.0) :=
        // access to in parameters is allowed
        @dynamic { return fx_distance(p_origin, value) <= p_maxDistance; };
        // mw_closeTo(c,d) matches all values of type Coordinate
// which have maximum distance of d from Coordinate c

    external function @deterministic fx_isPrime(integer p_x) return boolean;
    :
    p.receive(@dynamic fx_isPrime)
    // is the same as p.receive(integer:@dynamic { return fx_isPrime(value) })
```

5.2 Templates with variable bindings

5.2.0 General

The possibilities to retrieve field values of received messages and signatures in ETSI ES 201 873-1 [1] are restricted and cumbersome. To overcome this situation, this clause implements the definition of templates with variable bindings that store the actual value matched by a template instance if the matching of all containing templates is successful. This feature is implemented by using the "->" symbol for denoting a value assignment to a variable or an **out** parameter in the scope of a template definition in case of a successful template matching. Such a value assignment can syntactically be specified at all places where a template matching mechanism or a template reference is used in a template definition or an inline template.

5.2.1 Value retrieval from matching

In case of a successful template match, the value which matches the template can be assigned to a variable. This can be specified by using the "->" symbol.

Syntactical Structure

TemplateInstance **"->"** *VariableRef*

Semantic Description

If a template successfully matches a value and contains a value retrieval assignment for a *TemplateInstance*, then, if during the matching process that *TemplateInstance* and all its containing *TemplateInstances* contribute to the successful template match, the part of the value the *TemplateInstance* was matching is assigned to the *VariableRef* referenced in the value retrieval assignment.

EXAMPLE:

```
template integer mw_t1 := (?, (1..3) -> v); // mw_t1 always matches, but if the (1..3) does not
                          // contribute to the successful match
                          // then v is not assigned a value
template integer mw_t2 := ((1..3) -> v_small, (3..5) -> v_big)
 // matching mw_t2 to number in (1..3) will cause only v_small to be assigned,
 // matching it to number in (4..5) will cause only v_big to be assigned (preference of (1..3))
```

Restrictions

Additional to the restrictions given in clause 5.2.3, the following restrictions apply:

a) If *TempateInstance* describes the matching of a mandatory element in a template definition, *VariableRef* shall refer to a variable of the same type as the mandatory element.

b) If *TempateInstance* describes the matching of an optional element in a template definition, *VariableRef* shall refer to a template variable of the same type as the optional element. In case of a successful matching, the matching value or omit shall be assigned to the template variable denoted by *VariableRef*.

c) Value retrieval shall not be used in special places as described in clause 16.1.4 of ETSI ES 201 873-1 [1] with the exception of the matching parts of receiving operations. If value retrieval is used in the matching part of a receiving operation, the assignment to the variables of all the templates is done once the whole matching part matches and all other conditions for the selection of the alternative are met.

d) Value retrieval shall not be applied to templates of set of types or any of their direct or indirect elements, elements of a permutation matching mechanism or any of their direct or indirect elements.

NOTE: TTCN-3 makes no assumptions about the value of the variable or template variable denoted by *VariableRef* My in case of an unsuccessful match.

5.2.2 Declaring out parameters for template definitions

The retrieval of field values in case of successful matching from a structured value can be specified by means of **`out`** parameters of template definitions.

Syntactical Structure

The syntactical structure for global and local template definitions does not change:

template [*restriction*] [**@fuzzy**] *Type TemplateIdentifier* ["(" *TemplateFormalParList* ")"]
[**modifies** *TemplateRef*] ":=" *TemplateBody*

Only the static semantics restriction for formal template parameters change in such a way that formal template parameters shall evaluate to **`in`** or **`out`** parameters.

Semantic Description

The semantics of the value retrieval using templates with `out` parameters can be described by means of the **`match`** operation:

```
// Given the record type definition
type record MyRecordType {
    integer     field1,
    boolean     field2
}

// the constant definition
const MyRecordType c_myRecord := {
    field1 := 7,
    field2 := true
}

// the template definition
template MyRecordType mw_myTemplate1 := {
    field1 := 7,
    field2 := ?
}

// the same template definition with an out parameter matchinstring
template MyRecordType mw_myTemplate2 (out boolean p_matchingBool):= {
    field1 := 7,
    field2 := ? -> p_matchingBool
}

// and the variable declarations
var boolean v_a, v_b;
```

```
    // then the effect of
    v_a := match (c_myRecord, mw_myTemplate2(v_b));

    // is identical to
    if (match (c_myRecord, mw_myTemplate1) {
        v_a := true;
        v_b := c_myRecord.field2        // i.e., true
    }
    else {
        v_a:= false;
    }
```

Restrictions

Additional to the restrictions given in clause 5.2.3, the following restrictions apply:

a) *TempateInstance* with `out` parameters shall not be used as templates of `set of` types or any of their direct or indirect elements.

b) *TempateInstance* with `out` parameters shall not be used as elements of a permutation matching mechanism or any of their direct or indirect elements. *TempateInstance* with `out` parameters shall not be used in special places as described in clause 16.1.4 of ETSI ES 201 873-1 [1] with the exception of the matching parts of receiving operations. If *TempateInstance* with `out` parameters are used in the matching part of a receiving operation, the assignment to the actual `out` parameter variables of all the templates is done only once the whole matching part matches and all other conditions for the selection of the alternative are met.

c) Every formal `out` parameter of a template with `out` parameters shall be referenced at most once inside the template body.

NOTE: In certain cases a template may match without assigning a value to a declared `out` parameter. In such cases, the boundness of the actual `out` parameter has to be checked even after a successful match before using it.

EXAMPLE 1:

```
type union Tree {
    integer leaf,
    Branch branch
}
type record Branch {
    Tree left optional,
    Tree right optional
}

template Branch mw_branch(out omit Tree p_left, out omit Tree p_right) := {
    left := * -> p_left,
    right := * -> p_right
}
template Tree mw_treebranch(out omit Tree p_left, out omit Tree p_right) :=
    { branch := mw_branch(p_left, p_right) }
template Tree mw_leaf(out integer p_value, in template integer p_expected := ?) :=
    { leaf := p_expected -> p_value }

// compute the sum of absolute values of all leafs in the given tree
function f_absSum(omit Tree p_tree) return integer {
    var omit Tree v_left, v_right;
    var integer v_value;
    if (ispresent(p_tree)) {
        select (p_tree) {
            case (mw_treebranch(v_left, v_right)) {
                    return f_absSum(v_left) + f_absSum(v_right);
                }
            case (mw_leaf(v_value, (0 .. infinity)) { return v_value; }
            case (mw_leaf(v_value) { return -v_value; }
        }
    }
    else {
        return 0; }
}
```

EXAMPLE 2:

```
var Tree v_tree;
…
template Tree mw_anyTree(out integer p_value,
                                 out omit Tree p_left, out omit Tree p_right) :=
    (mw_leaf(p_value), mw_treebranch(p_left, p_right));

// extraction of leaf or branch from v_tree (any value, if present)
if (match(v_tree, mw_anyTree(v_value, v_left, v_right)) {
    if (isbound(v_value)) {
        // v_value might not have been assigned => needs check
    }
    else if (isbound(v_left)) {
        // v_left and v_right might not have been assigned
        // => needs check
    }
}
```

EXAMPLE 3: Receiving with multiple out parameter templates.

```
signature S(out Tree p_tree) return Tree;

alt {
    [match(v_tree, mw_leaf(v_value)] p.receive { }
    // NOT allowed to use template with out parameter in alt guard
[] p.getreply(S:{ mw_leaf(v_value) } value mw_treebranch(v_left, v_right)) {
    // if either mw_leaf or mw_treebranch does not match, v_value, v_left and v_right
    // all remain unchanged
    // it is equivalent with the following:
    // [] p.getreply(S:{mw_leaf(v_value1)} value mw_treebranch(v_left1, v_right1)) {
    //  v_left := v_left1;
    //  v_right := v_right1;
    //   v_value := v_value1;
    // }
    // where v_value1, v_left1 and v_right1 are all internal unbound variables of the
    // appropriate ty
```

5.2.3 Matching template parameters and variables

Matching template parameters and variables can be safely used for value retrieval during matching. Such parameters and variables include the `@match` modifier in their declaration.

NOTE: Storing template instances which are binding local variable references to formal out parameters or directly to value redirection as normal variables or parameters could lead to situations where the content of the variable is used though the variables that are bound to its content are not in existence anymore. The notion of matching templates allows to distinguish normal templates from templates (potentially) incorporating value retrieval and eliminates the issue with non-existent target variables.

Syntactical Structure

```
var template [ restriction ]
[ (@lazy | @fuzzy) [ @deterministic ] ]
[ @match ]
  Type { VarIdentifier [ ArrayDef ] ":=" TemplateBody }+ [ ";" ]

[ in | inout | out ] template [ Restriction ]
[ ( @lazy | @fuzzy ) [@deterministic] ]
[ @match ]
Type ValueParIdentifier [ArrayDef] [ ":=" ( TemplateInstance | "-" ) ]
```

Semantic Description

Templates are categorized as matching templates if they have at least one of the following properties:

a) They are a matching mechanism with a value retrieval.

b) They are an instance of a template with at least one out parameter.

c) They are a reference to a template variable declared with the @match modifier.

d) They are a reference to a formal parameter declared with the @match modifier.

e) They contain a matching template as one of their fields or elements or part thereof.

Matching templates can be only used in assignments, as actual parameters, in template body notations, in the match operation, in the matching parts of receiving operations and case clauses of the select operation.

Restrictions

a) When used in a formal template parameter declaration, the `@match` modifier is allowed to be present only in *in* parameter declaration. Using the `@match` modifier in *out* or *inout* parameter declaration shall cause an error.

b) If a matching template is assigned to a variable, the variable shall have the `@match` modifier.

c) If a matching template is used as an actual parameter, the corresponding formal parameter shall have the `@match` modifier.

d) Variables with the `@match` modifier shall never be partially initialized. Referencing fields or elements both on the left hand side and on the right hand side of such variables is not allowed.

e) Actual parameters of formal parameters with the `@match` modifier shall be completely initialized.

f) Matching templates shall not be present in a return or raise statement.

g) Matching templates shall not be present in sending operations.

5.3 Additional logical operators for combining matching mechanisms

5.3.0 General

This clause defines the additional logical operators conjunction, implication, exclusion and disjunction for matching mechanisms. Their usage allows the combination matching mechanisms for advanced matching.

5.3.1 Conjunction

The conjunction matching mechanism is used when a value shall fulfil several distinct conditions.

Syntactical Structure

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

Semantic Description

The conjunction matching mechanism can be used for matching values of all types.

A template specified by the conjunction matching mechanism matches the corresponding value if and only if the value matches all templates listed in the template list.

Besides specifying individual templates, it is possible to add all elements of an existing **`record of`** or **`set of`** template into a conjunction matching mechanism using an **`all from`** clause.

Restrictions

a) The type of the conjunction matching mechanism and the member type of the template in the **`all from`** clause shall be compatible.

b) The template in the **`all from`** clause as a whole shall not resolve into a matching mechanism (i.e. its elements may contain any of the matching mechanisms or matching attributes with the exception of those described in the following restriction).

c) Individual fields of the template in the **all from** clause shall not resolve to any of the following matching mechanisms: *AnyElementsOrNone*, permutation, disjunction or repetition.

d) Each value or template in the list shall be compatible with the type declared for the template specified by the conjunction matching mechanism.

e) Referencing a sub-field within a template to which the conjunction matching mechanism is assigned (both at the right and left hand side of an assignment) shall cause an error.

EXAMPLE:

```
// external function returning true for prime numbers
external function f_isprime (integer p_par) return boolean;
// the template matches prime numbers greater than 100
template integer mw_myTemplate:= conjunct((100 .. infinity), @dynamic f_isprime);

type record of integer MySequenceOfType;
template MySequenceOfType mw_myTemplate2 := {1,2,3};
// mw_conjunctTemplate2 does not match any value as no value can be 1, 2 and 3 at the same time.
template integer mw_conjunctTemplate2 := conjunct(all from mw_myTemplate2);

template MySequenceOfType mw_myTemplate3 := {2,2,2};
// mw_conjunctTemplate3 matches the single value 2
template integer mw_conjunctTemplate3 := conjunct(all from mw_myTemplate3);
```

5.3.2 Implication

The implication matching mechanism is used when a match is checked only if specified conditions are met.

Syntactical Structure

TemplateInstance **implies** *TemplateInstance*

Semantic Description

The implication matching mechanism is composed of two templates separated by an **implies** keyword. The first template is called *precondition* and the second one is called the *implied template*. The implication matching mechanism can be used for matching values of all types.

A template specified by the implication matching mechanism matches a value if and only if the value does not match the precondition or if the value matches both the precondition and implied template.

Restrictions

a) Both templates used in the implication matching mechanism shall be compatible to the type declared for the template specified by the implication matching mechanism.

b) Any matching attribute at the end of the implication matching mechanism is related to the whole template (i.e. not only to the implied template).

c) Referencing a sub-field within a template specified by the implication matching mechanism (both at the right and left hand side of an assignment) shall cause an error.

EXAMPLE:

```
// the template matches strings with 5 or more characters ending with "abc" and all shorter
// strings (i.e. strings with length 0..4)
template charstring mw_implies1 := ? length (5 .. infinity) implies pattern "*abc";

// the following template matches all strings with length 0..4 and all strings with length
// 8..10 ending with "abc".
template charstring mw_implies2 :=
    ? length (5 .. infinity) implies (pattern "*abc" length (8..10));

// the following template matches all strings with length 2..4 and all strings with length
// 5..10 ending with "abc" because the length attribute is applied to the whole implication.
template charstring mw_implies3 :=
    ? length (5 .. infinity) implies pattern "*abc" length (2..10);
```

5.3.3 Exclusion

The exclusion matching mechanism is used when a general matching rule is to be restricted by an exception.

Syntactical Structure

TemplateInstance **except** *TemplateInstance*

Semantic Description

The first template in exclusion matching mechanism is called *general template* and the second one is called *excluded template*. The exclusion matching mechanism can be used for matching values of all types.

A template that is specified by the exclusion matching mechanism matches a value if and only if the value matches the general template, but does not match the excluded template.

Restrictions

a) Both templates used in the exclusion matching mechanism shall be compatible to the type declared for the template specified by the exclusion matching mechanism.

b) Any matching attribute at the end of the exclusion matching mechanism is related to the whole matching mechanism (i.e. not only to the excluded template).

c) Referencing a sub-field within a template specified by the exclusion matching mechanism (both at the right and left hand side of an assignment) shall cause an error.

EXAMPLE:

```
// the template matches all integer in the range 1..100 except 48 and 64
template integer mw_int := (1..100) except (48, 64);

// the following template matches all strings with length 0..10 except strings starting
// with "abc" that contain 3..5 characters.
template charstring mw_str := ? length (0 .. 10) except (pattern "abc*" length (3..5));

// the following template matches all strings with length 0..10 starting with "abc", but not
// ending with "xyz".
template charstring mw_str2 := pattern "abc*" except pattern "*xyz" length (0..10);
```

5.3.4 Disjunction

The disjunction matching mechanism specifies one or more alternatives for matching multiple elements of **`record of`**, **`set of`** or array.

Syntactical Structure

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

Semantic Description

The disjunction mechanism is used only inside values of type **`record of`**, **`set of`** or array. The template instances specified in the disjunction matching mechanism are called *disjunction alternatives*. It provides a successful match, if and only if one of the disjunction alternatives matches the maximal subset of still unmatched elements inside the value that still allows the containing template to match the whole value. And in case of **`record of`** and array values the matched subset shall be a consecutive sub-sequence beginning after the end of the sub-sequence that was matched by the preceding inside-values templates in the containing template.

The disjunction alternatives are matched in the order of their specification inside the disjunction matching mechanism (i.e. from left to right) so that the leftmost alternative that fulfils the above condition contributes to the successful match while the following disjunction alternatives are ignored.

Besides specifying all individual values, it is possible to add all elements of an existing **`record of`** or **`set of`** template into disjunction using an **`all from`** clause.

Restrictions

a) The type of templates used in the disjunction matching mechanism that are not prefixed with the **all from** clause and the type of the **record of**, **set of** or array containing the disjunction matching mechanism shall be compatible.

b) The type of the **record of**, **set of** or array containing the disjunction matching mechanism and the member type of the template in the **all from** clause shall be compatible.

c) The template in the **all from** clause as a whole shall not resolve into a matching mechanism (i.e. its elements may contain any of the matching mechanisms or matching attributes with the exception of those described in the following restriction).

d) Individual fields of the template in the **all from** clause shall not resolve to any of the following matching mechanisms: *AnyElementsOrNone*, permutation, disjunction or repetition.

e) Referencing an element located within the disjunction matching mechanism both at the left or right hand side of an assignment shall cause an error.

f) In case the templates within the disjunction matching mechanism can match values of different length, referencing an element following the disjunction matching mechanism in the containing template shall cause an error.

g) If the disjunction mechanism contains items that all match only values of the same fixed length or has a fixed length attribute attached to it and all other required conditions are met, the items following the disjunction in the containing template may be referenced at the left or right hand side of an assignment. The index of the first item following the disjunction is equal to the index of the disjunction matching mechanism increased by its fixed length.

EXAMPLE:

```
type record of integer RoI;
template RoI mw_a := { 1, 2 }
template RoI mw_b := { 1, *, 4 }
template RoI mw_disjunction := {0, disjunct( mw_a, mw_b, ? length(2..4) ), 10, *};
// the previous template matches the same values as the following template:
template RoI mw_list := ({0, 1, 2, 10, *}, {0, 1, *, 4, 10, *}, {0, * length(2..4), 10, *});
```

5.4 Repetition

5.4.1 General

Repetition is a specific symbol used inside **record of**, array, **bitstring**, **hexstring** or **octetstring** templates to match repeated sequences of items.

5.4.2 Repetition in record of and array

In **record of** and array templates, the repetition symbol is used to match repetitions of sub-sequence templates inside values of that type.

Syntactical Structure

TemplateInstance "#" "(" [SingleExpression] ["," [SingleExpression]] ")"

Semantic Description

The repetition matching mechanism shall be used only inside values. It consists of a *TemplateInstance* called the *repeated template* matching a sequence of elements of fixed length followed by a declaration of the number of required consecutive matches of that repeated template. It shall match if, and only if the specified number of consecutive sub-sequences of the value to be matched, following the elements already matched by preceding inside value matching mechanisms, are matching the *repeated template*. The number of required matches shall be specified by using one of the following syntaxes: "#(n, m)", "#(n,)", "#(, m)", "#(n)", "#n", "#(,)", "#()".

The form "#(n, m)" specifies that at least *n* sub-sequence but not more than *m* sub-sequences shall match the repeated template.

The form "#(n,)" specifies that at least *n* sub-sequences shall match the repeated template. The maximum number of matches is not restricted in this case.

The form "#(, m)" indicates that at most *m* sub-sequences shall match the repeated template. The minimum number of matches is not set, thus 0..*m* positive matches are allowed.

The single value form "#(n)" requires that precisely *n* sub-sequences shall be matched by the repeated template to produce a successful match.

The forms "#(,)" and "#()" are shorthand notations for "#(0,)", i.e. any number of sub-sequences shall match the repeated template.

The repetition matches the longest number of elements possible that still allows the containing template to match the whole value.

The repeated template can reference the current iteration of the repetition by using the special keyword @index. This can be used to assign parts of each matches sub-sequence of the value to different variables when the repeated template uses value retrieval or has out parameters (see clause <Value retrieval>).

Restrictions

a) The type of the repeated template and the type of the containing template shall be compatible.

b) The expressions used for specifying the number of required sub-sequences shall resolve to a non-negative integer value.

c) In case there are two expressions specifying the number or required sub-sequences, the second value shall be greater or equal to the first one.

d) Referencing an element located within the repeated template shall cause an error.

e) In case any other notation than #(n) or #(n,m) where *n* is equal to *m* is used (i.e. when the number of required sub-sequences is not a fixed value), referencing an element following the repetition shall cause an error.

f) If the repetition uses the #(n) notation or #(n,m) notation where *n* and *m* are equal and all other required conditions are met, the template items following the repetition matching mechanism may be referenced at the left or right hand side of an assignment. The index of the first item following the repetition is equal to the index of the repetition increased by *n* * *l* where *l* is the fixed length of the repeated template.

g) The repeated template shall match only sub-sequences of a single fixed length, so that every matched sub-sequence will have the same length.

NOTE: To specify a fixed length repeated template, care should be taken when using *AnyElementsOrNone*, *AnyElement*, disjunction and repetition matching mechanisms. Normally, they or their sub-templates have to be restricted with a length attribute of fixed length.

EXAMPLE 1:

```
type record Name {
    charstring givenName,
    charstring surname
}
type record of Name RoN;
template RoN mw_a := { { { givenName := "John", surname := ?} } #(2, 5) }
    // the template matches values containing 2..5 elements; the givenName field of each
    // element has to be equal to "John"
template RoN mw_b := { { { givenName := ?, surname := "Doe"} } #() }
    // the template matches only values with the surname field of each element equal to "Doe"
    // and values with no elements
```

EXAMPLE 2: Using the @index operator to assign matched values to variables.

```
var charstring v_name[5];
template RoN mw_c(out charstring p_name) :=
    { { givenName := "John", surname := ? -> p_name } }
// the following two template are semantically equivalent and will assign the surname field
// of each matched template in each iteration of the repetition to a different element
// of the variable v_name
template RoN mw_d := { { { givenName := "John", surname := ? -> v_name[@index]} } #(2, 5) }
template RoN mw_e := { mw_c(v_name[@index]) #(2, 5) }
```

EXAMPLE 3: Using the @index operator in nested repetitions.

```
type record of RoN RRoN;
type record of charstring RoC;
type record of RoC RRoC;
var RRoC v_nestedNames;
// the inner repletion has to be restricted to a fixed length
template RoN mv_f(out RoC p_names) := { mw_c(p_names[@index]) #(5) }
// because we can only access the current index, each repetition layer can be
// dealt with by adding a new template
template RRoN mv_nestedRepetition := { mv_f(v_nestedNames[@index])# () }
```

5.4.3 Repetition in string

Inside templates of **`bitstring`**, **`hexstring`** and **`octetstring`** types, the repetition symbol is used to match repeated occurrence of a string element.

Syntactical Structure

"#"(num | (" (" [number]["," [number]] ")"))

The repetition symbol shall be used only inside values. It is used to specify that a preceding symbol of a binary string template (*AnyElement* or string element) should be matched a number of times. The following syntaxes are available: "#(n, m)", "#(n,)", "#(, m)", "#(n)", "#n", "#(,)" or "#()".

The form "#(n, m)" specifies that the preceding symbol shall be matched at least *n* times but not more than *m* times.

The metacharacter postfix "#(n,)" specifies that the preceding symbol shall be matched at least *n* times while "#(, m)" indicates that the preceding expression shall be matched not more than *m* times.

Metacharacters (postfixes) "#(n)" and "#n" specify that the preceding expression shall be matched exactly n times (they are equivalent to "#(n, n)"). In the form "#n", *n* shall be a single digit.

The forms "#(,)" and "#()" are shorthand notations for "#(0,)", i.e. matches the preceding expression any number of times.

Restrictions

a) Using repetition as the first item in the string shall cause an error.

b) Adding another repetition to a symbol that already has repetition attached shall cause an error.

c) Applying repetition to *AnyElementsOrNone* has no effect.

EXAMPLE:

```
template bitstring mw_repe1 := '1?#(3,5)'B;
       // matches bitstrings whose first bit is equal to one, followed by 3..5 other bits


template octetstring mw_repe2 := 'AB#3'O;
       // matches an octestring 'ABABAB'O
```

5.4.4 Modifications to ETSI ES 201 873-1 [1], clause 15.11 (Concatenating templates of string and list types)

5.4.4.0 General

The modifications to ETSI ES 201 873-1 [1], clause 15.11 (Concatenating templates of string and list types) shall be done in two steps. Step 2 (described in clause 5.4.4.2) shall be performed on the result of Step 1 (described in clause 5.4.4.1).

5.4.4.1 Step 1 of modifications to ETSI ES 201 873-1 [1], clause 15.11

The second and third paragraph of clause 15.11 of ETSI ES 201 873-1 [1] are replaced with the following ones:

The single templates of binary string types shall evaluate only to the matching mechanisms specific value, combined template, *AnyValue* with or without a length attribute or *AnyValueOrNone* with a length attribute.

The concatenation of templates of binary string types results in the sequential concatenation of the single templates from left to right. All matching symbols *AnyValue* and *AnyValueOrNone* are replaced by *AnyElement* (possibly followed by a repetition symbol) or *AnyElementsOrNone* matching symbols according to the table 5.1 before concatenation.

Table 5.1: Transformation of binary string templates before concatenation

Concatenation operand	Transformed bitstring	Transformed hexstring	Transformed octetstring
?, ? length(0..infinity) or * length(0..infinity)	'*'B	'*'H	'*'O
? length(0) or * length(0)	''B	''H	''O
? length(1) or * length(1)	'?'B	'?'H	'?'O
? length(*n*) or * length(*n*)	'?#(*n*)'B	'?#(*n*)'H	'?#(*n*)'O
? length(*n*, infinity) or * length(n, infinity)	'?#(*n*,)'B	'?#(*n*,)'H	'?#(*n*,)'O
? length(*n*, *m*) or * length(*n*, *m*)	'?#(*n*,*m*)'B	'?#(*n*,*m*)'H	'?#(*n*,*m*)'O

5.4.4.2 Step 2 of modifications to ETSI ES 201 873-1 [1], clause 15.11

The result of the modifications described in clause 5.4.4.1 shall be amended according to the following rules:

In addition to the matching symbols specified in ETSI ES 201 873-1 [1] clause 15.11, and clauses 5.4.2 and 5.4.3 of the present document, it is allowed to use other types of matching symbols in concatenation of templates of string, **`record of`** and array types.

A template that is a result of concatenation produces a successful match if a there is at least one possible way of splitting the value being matched into *n* consecutive subsections where *n* is equal to the number of concatenated templates so that all individual operands of the concatenated template produce a successful match when matching the split subsections. Matching of subsections is performed in the textual order; an operand of the concatenated template shall match a subsection at the same ordinal position.

Restrictions

a) The matching symbols used in concatenation of templates of string, **`record of`** and array types shall be either one of the matching symbols specified in ETSI ES 201 873-1 [1] clause 15.11 and clause 5.4.2 of the present document or it shall obey the present template restriction.

EXAMPLE:

```
template octetstring mw_t1 := ('1234'O, '5678'O) & complement('*ABCD'O);
      // matches any octetstring that starts either with '1234'O or '5678'O and doesn’t end with
      // 'ABCD'O:

   template charstring mw_activity := ("John", "Jack", "Frank") & ("played " &
      ("football", "basketball", "hockey", "baseball"), "ran " & ("fast", "slowly"));
   log(match("John played basketball", mw_activity)); // produces true
```

```
log(match("Jack ran slowly", mw_activity)); // produces true
log(match("Frank played fast", mw_activity)); // produces false, because "played" is not
    // concatenated with "fast"

type record of charstring RoCS;
template RoCS mw_rocs := ? & ({"Paris", "London"}, {"New York", "Beijing"}) & ?;
    // matches any record of value that contains "Paris" followed by "London" or "New York"
    // followed by "Beijing"
```

5.4.5 Modifications to ETSI ES 201 873-6 [4]

5.4.5.1 Changes to clause 7.2.2.3.1 (The abstract data type `MatchingMechanism`)

`TciMatchingTypeType getMatchingType()`

Returns the matching mechanism type. A value of `TciMatchingTypeType` can have one of the following constants: `TEMPLATE_LIST, COMPLEMENTED_LIST, ANY_VALUE, ANY_VALUE_OR_NONE, VALUE_RANGE, SUBSET, SUPERSET, ANY_ELEMENT, ANY_ELEMENTS_OR_NONE, PATTERN, MATCH_DECODED_CONTENT, OMIT_TEMPLATE, CONCATENATION.`

5.4.5.2 Changes to clause 7.2.2.3.2 (The abstract data type `MatchingList`)

The abstract data type `MatchingList` is used to represent matching mechanisms that contain a list of items of the same type: template list, complemented template list, SubSet, SuperSet and concatenation.

5.4.5.3 Changes to clause 8.3.2.17 (TciMatchingTypeType)

`TciMatchingTypeType` is mapped to the following interface:

```
// TCI IDL TciTypeClassType
package org.etsi.ttcn.tci;
public interface TciMatchingType {
    public final static int TEMPLATE_LIST           = 0 ;
    public final static int COMPLEMENTED_LIST       = 1 ;
    public final static int ANY_VALUE               = 2 ;
    public final static int ANY_VALUE_OR_NONE       = 3 ;
    public final static int VALUE_RANGE         = 4 ;
    public final static int SUBSET                  = 5 ;
    public final static int SUPERSET                = 6 ;
    public final static int ANY_ELEMENT         = 7 ;
    public final static int ANY_ELEMENTS_OR_NONE    = 8 ;
    public final static int PATTERN             = 9 ;
    public final static int MATCH_DECODED_CONTENT   = 10 ;
    public final static int OMIT_TEMPLATE           = 11 ;
    public final static int CONCATENATION           = 12 ;
}
```

5.4.5.4 Changes to clause 8.3.3.1 (Type)

- `newTemplate`

  Returns a freshly created matching mechanism of this type. The `matchingType` parameter determines what kind of matching mechanism will be created and it shall be one of the following constants: `TEMPLATE_LIST, COMPLEMENTED_LIST, ANY_VALUE, ANY_VALUE_OR_NONE, VALUE_RANGE, SUBSET, SUPERSET, ANY_ELEMENT, ANY_ELEMENTS_OR_NONE, PATTERN, DECODED_MATCH, CONCATENATION`. If the created matching mechanism contains additional data properties, these properties are uninitialized in the created matching mechanism.

5.4.5.5 Changes to clause 9.5 (Data)

Table 5.2

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

TciModuleParameterType	`typedef struct TciModuleParameterType`
	`{`
	`  QualifiedName parName;`
	`  Value         defaultValue;`
	`} TciModuleParameterType;`	

TciModuleParameterListType	`typedef struct TciModuleParameterListType`
	`{`
	`  long int               length;`
	`  TciModuleParameterType *modParList;`
	`} TciModuleParameterListType;`	

TciParameterType	`typedef struct TciParameterType`
	`{`
	`  String   parName;`
	`  TciParameterPassingModeType parPassMode;`
	`  Value                      parValue;`
	`} TciParameterType;`	

TciParameterPassingModeType	`typedef enum`
	`{`
	`  TCI_IN_PAR    = 0,`
	`  TCI_INOUT_PAR = 1,`
	`  TCI_OUT_PAR   = 2`
	`} TciParameterPassingModeType;`	

TciParameterListType	`typedef struct TciParameterListType`
	`{`
	`  long int          length;`
	`  TciParameterType *parList;`
	`} TciParameterListType;`
		length 0 shall be interpreted as "empty list".

TciParameterTypeListType	`typedef struct TciParameterTypeListType`
	`{`
	`  long int  length;`
	`  TciParameterTypeType *parList;`
	`} TciParameterTypeListType;`
		length 0 shall be interpreted as "empty list".

TciParameterTypeType	`typedef struct TciParameterTypeType {`
	`    String parName;`
	`    Type parameterType;`
	`    TciParameterPassingModeType mode;`
	`} TciParameterTypeType;`	

TciTestCaseIdListType	`typedef struct TciTestCaseIdListType`
	`{`
	`  long int       length;`
	`  QualifiedName *idList;`
	`} TciTestCaseIdListType;`
		length 0 shall be interpreted as "empty list".

TciTypeClassType	`typedef enum`
	`{`
	`  TCI_ADDRESS_TYPE = 0,`
	`  TCI_ANYTYPE_TYPE = 1,`
	`  TCI_BITSTRING_TYPE = 2,`
	`  TCI_BOOLEAN_TYPE = 3,`
	`  TCI_CHARSTRING_TYPE = 5,`
	`  TCI_COMPONENT_TYPE = 6,`
	`  TCI_ENUMERATED_TYPE = 7,`
	`  TCI_FLOAT_TYPE = 8,`
	`  TCI_HEXSTRING_TYPE = 9,`
	`  TCI_INTEGER_TYPE = 10,`
	`  TCI_OCTETSTRING_TYPE = 12,`
	`  TCI_RECORD_TYPE = 13,`
	`  TCI_RECORD_OF_TYPE = 14,`
	`  TCI_ARRAY_TYPE = 15,`
	`  TCI_SET_TYPE = 16,`
	`  TCI_SET_OF_TYPE = 17,`
	`  TCI_UNION_TYPE = 18,`
	`  TCI_UNIVERSAL_CHARSTRING_TYPE = 20,`
	`  TCI_VERDICT_TYPE = 21`
	`  TCI_DEFAULT_TYPE = 22,`
	`  TCI_PORT_TYPE = 23,`
	`  TCI_TIMER_TYPE = 24`
	`} TciTypeClassType;`	

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

TciTestComponentKindType	typedef enum
	{
	  TCI_CTRL_COMP,
	  TCI_MTC_COMP,
	  TCI_PTC_COMP,
	  TCI_SYS_COMP,
	  TCI_ALIVE_COMP
	} TciTestComponentKindType;	

TciBehaviourIdType	QualifiedName	

TciValueDifference	typedef struct TciValueDifference
	{
	  Value val;
	  TciValueTemplate tmpl;
	  String desc;
	} TciValueDifference;	

TciValueDifferenceList	typedef struct TciValueDifferenceList
	{
	  long int length;
	  TciValueDifference* diffList;
	} TciValueDifferenceList;
		length 0 shall be interpreted as "empty list".

TciMatchingTypeType	typedef enum
	{
	  TCI_TEMPLATE_LIST = 0,
	  TCI_COMPLEMENTED_LIST = 1,
	  TCI_ANY_VALUE = 2,
	  TCI_ANY_VALUE_OR_NONE = 3,
	  TCI_VALUE_RANGE = 4,
	  TCI_SUBSET = 5,
	  TCI_SUPERSET = 6,
	  TCI_ANY_ELEMENT = 7,
	  TCI_ANY_ELEMENTS_OR_NONE = 8,
	  TCI_PATTERN = 9,
	  TCI_MATCH_DECODED_CONTENT = 10,
	  TCI_OMIT_TEMPLATE = 11,
	  TCI_CONCATENATION = 12
	} TciMatchingTypeType;	

LengthRestriction	typedef struct TciLengthRestriction
	{
	  unsigned long int lowerBoundary;
	  unsigned long int upperBoundary;
	  Boolean isUpperBoundaryInfinity;
	} TciLengthRestriction;	

Permutation	typedef struct TciPermutation
	{
	  unsigned long int startPosition;
	  unsigned long int length;
	  } TciPermutation;	

RangeBoundary	typedef struct TciRangeBoundary
	{
	  Value boundary;
	  Boolean isInclusive;
	  Boolean isInfinity;
	} TciRangeBoundary;	

5.4.5.6 Changes to clause 10.5.2.16 (TciMatchingTypeType)

Defines the matching template type. It is mapped to an enumeration:

```
typedef enum
{
  TCI_TEMPLATE_LIST = 0,
  TCI_COMPLEMENTED_LIST = 1,
  TCI_ANY_VALUE = 2,
  TCI_ANY_VALUE_OR_NONE = 3,
  TCI_VALUE_RANGE = 4,
  TCI_SUBSET = 5,
  TCI_SUPERSET = 6,
  TCI_ANY_ELEMENT = 7,
  TCI_ANY_ELEMENTS_OR_NONE = 8,
  TCI_PATTERN = 9,
  TCI_MATCH_DECODED_CONTENT = 10,
  TCI_OMIT_TEMPLATE = 11,
  TCI_CONCATENAION = 12
} TciMatchingType;
```

5.4.5.7 Changes to clause 10.5.3.19 (MatchingList)

Represents the following TTCN-3 matching mechanisms: template list, complemented template list, subset, superset, concatenation. It is mapped to the following pure virtual class.

5.4.5.8 Changes to clause 11.3.3.25 (MatchingMechanism)

`MatchingMechanism` is mapped into a sub-element of a typed value element. The sub-element is based on the complex type specified below:

```
<xsd:complexType name="MatchingSymbol">
    <xsd:choice>
        <xsd:element name="any_value" type="SimpleTypes:TEmpty"/>
        <xsd:element name="any_value_or_none" type="SimpleTypes:TEmpty"/>
        <xsd:element name="any_element " type="SimpleTypes:TEmpty"/>
        <xsd:element name="any_element_or_none" type="SimpleTypes:TEmpty"/>
        <xsd:element name="range" type="Templates:Range"/>
        <xsd:element name="list" type="Templates:MatchingList"/>
        <xsd:element name="complement" type="Templates:MatchingList"/>
        <xsd:element name="subset" type="Templates:MatchingList"/>
        <xsd:element name="superset" type="Templates:MatchingList"/>
        <xsd:element name="permutation" type="Templates:MatchingList"/>
        <xsd:element name="decmatch" type="Templates:DecMatch"/>
        <xsd:element name="concatenation" type="Templates:MatchingList"/>
    </xsd:choice>
</xsd:complexType>
```

Choice of Elements:

- `any_value` The *AnyValue* matching symbol.
- `any_value_or_none` The *AnyValueOrNone* matching symbol.
- `any_element` The *AnyElement* matching symbol.
- `any_element_or_none` The *AnyElementOrNone* matching symbol.
- `range` A range template.
- `list` A template list.
- `complement` A complemented template list.
- `subset` A subset template.
- `superset` A superset template.
- `permutation` A permutation.
- `pattern` A pattern.
- `decmatch` A *MatchDecodedContent* matching mechanism.
- concatenation A concatenation template for a record of type.

Attributes:

- The same attributes as those of Value.

5.4.5.9 Changes to clause 12.4.2.16 (TciMatchingTypeType)

`TciMatchingTypeType` is mapped to the following enumeration:

```
public enum TciMatchingType
{
    TemplateList = 0,
    ComplementedList = 1,
    AnyValue = 2,
```

```
    AnyValueOrNone = 3,
    ValueRange = 4,
    Subset = 5,
    Superset = 6,
    AnyElement = 7,
    AnyElementsOrNone = 8,
    Pattern = 9,
    MatchDecodedContent = 10,
    OmitTemplate = 11,
    Concatenation = 12
}
```

5.4.5.10 Changes to clause B.4 (TCI-TL XML Schema for Templates)

```
    <xsd:complexType name="MatchingSymbol">
        <xsd:choice>
            <xsd:element name="any_value" type="SimpleTypes:TEmpty"/>
            <xsd:element name="any_value_or_none" type="SimpleTypes:TEmpty"/>
            <xsd:element name="any_element " type="SimpleTypes:TEmpty"/>
            <xsd:element name="any_element_or_none" type="SimpleTypes:TEmpty"/>
            <xsd:element name="range" type="Templates:Range"/>
            <xsd:element name="list" type="Templates:MatchingList"/>
            <xsd:element name="complement" type="Templates:MatchingList"/>
            <xsd:element name="subset" type="Templates:MatchingList"/>
            <xsd:element name="superset" type="Templates:MatchingList"/>
            <xsd:element name="permutation" type="Templates:MatchingList"/>
            <xsd:element name="pattern" type="Templates:Pattern"/>
            <xsd:element name="decmatch" type="Templates:DecMatch"/>
            <xsd:element name="concatenation" type="Templates:MatchingList"/>
        </xsd:choice>
    </xsd:complexType>
```

5.5 Equality operator for the omit symbol and templates with restriction omit

5.5.0 General

This clause removes restrictions for the use of the omit symbol and templates with restriction omit. It allows the usage of an omit symbol and templates with restriction omit as operands of the equality operator.

5.5.1 Modifications to ETSI ES 201 873-1 [1], clause 7 (Expressions)

Clause 7.1.3 Relational operators

Add the following text before the bullet points:

The **`omit`** keyword is allowed to be used in place of an operand for the equality (==) and non-equality (!=) operators, if the other operand is a reference and the referenced field is an **`optional`** field or a reference to a template declared with the **`omit`** restriction.

Add the following bullets to the list of bullet points:

- When the **`omit`** keyword is used in place of an operand for the equality operator, the equality check evaluates to **`true`** if and only if the other operand is set to **`omit`**, otherwise, it evaluates to **`false`**.
- When the **`omit`** keyword is used in place of an operand for the non-equality operator, the non-equality check evaluates to **`true`** if and only if the other operand is not set to **`omit`**, otherwise, it evaluates to **`false`**.

In EXAMPLE change the text as follows:

- Delete the following lines, shown in strike-through font:

~~`c_s1.a1 == omit;`~~
~~`// error, omit is neither a value nor a field reference`~~

- Insert the following lines before the line, next to the deleted ones:

```
c_s1.a1 == omit;
// returns false
c_s1.a2 == omit;
// returns true
c_u1.d1 == omit; // error, c_u1.d1 is not a reference to an optional field
omit != c_s1.a2;
// returns false
c_s1.a1 != omit;
// returns true

function compareToOmit(template(omit) integer i) return boolean {
    if (i == omit) { return true; } else { return false; }
    }
```

5.6 Encodable Values

5.6.0 General

Encodable values are all TTCN-3 entities that can be encoded. They are comprised of all values, value templates and encodable value templates. Encodable value templates are special templates which can be encoded but not be used as a value in any other way. Thus, the only operations allowed on encodable value templates are operations that involve encoding, i.e. the sending operations and the encvalue operations.

Restrictions

a) Encodable value templates shall not be assigned to variables or parameters of unrestricted templates or of templates with the present, value or omit restrictions.

b) Encodable value templates shall not be used as operands for expressions.

c) Encodable value templates shall not be used as value parameters.

d) Encodable value templates shall not be used in receive operations or in operations which require the value restriction.

5.6.1 The **encvalue** Template Restriction

Syntactical Structure

template "(" **encvalue** ")" | **template** "(" **encvalue** "," **omit** ")"

Semantic Description

To allow to differentiate between normal **value** restricted templates and encodable value templates, the template restriction (**encvalue)** shall be used. It is possible to define templates with that restriction as well as use the **template(encvalue)** modifier for the declaration of variables, formal parameters and return clauses. Additionally, to describe encodable templates that can be used in place of optional fields, the **(encvalue,omit)** restriction shall be used.

Templates with the (encvalue) restriction can be derived from templates with value or encvalue restriction by usage of modifies. Template with the (encvalue,omit) restriction can be derived from templates with value, encvalue or omit restriction by usage of modifies.

Any template that contains at least one part with an encvalue restriction also fulfils the encvalue restriction.

The following relationships between different kinds of templates exist:

- template(value) is subset of template(encvalue)
- template(encvalue) is subset of template(encvalue, omit)
- template(omit) is subset of template(encvalue, omit)
- template(encvalue) is not subset of unrestricted template
- template(encvalue, omit) is not subset of unrestricted template

Any template expression that is part of a subset of another template kind can be used also as the more general kind. Using a template with the **encvalue** restriction as a template without that restriction shall produce an error.

Examples

EXAMPLE 1:

```
template(encvalue) charstring msg_foobar := "foobar";

function f_encvalue(template(encvalue) charstring p_msg := msg_foobar)
return template(encvalue) charstring {
    var template(encvalue, omit) charstring v_msg := p_msg; // allowed
    return p_msg; // allowed
}
```

EXAMPLE 2:

```
template(omit) charstring msg_omit := "foobar";
template(encvalue, omit) charstring msg_encvalue_omit := msg_omit; // allowed

function f_encvalue(template(encvalue, omit) charstring p_msg := msg_encvalue_omit) // allowed
return template(encvalue) charstring {
    var template(encvalue) charstring v_msg := p_msg; // not allowed
    var template charstring v_msg2 := p_msg; // not allowed
    return p_msg; // not allowed
}
```

5.6.2 Encoding Mutation Annotation

5.6.2.0 Description

In encodable value templates, it is allowed to add a mutation annotation to parts of the template which is applied during encoding of the annotated part as a post-processing step to the original result produced by the encoder for that part.

Syntactical Structure

[*TemplateInstance*] (**@mutation** | **@mutation_o** | **@mutation_unichar** ["(" *StringEncoding* ")"]) *Expression*

Semantic Description

The family of mutation annotations **@mutation**, **@mutation_o** and **@mutation_unichar** are template expressions which conform to the **encvalue** template restriction.

If the *TemplateInstance* has the **omit** template restriction, then the resulting encodable value template has the restriction **(encvalue,omit)**. If the *TemplateInstance* has the **(value)** or **(encvalue)** restriction, the resulting encodable value template has the restriction **(encvalue)**. Encodable templates with the **(encvalue)** restriction can be safely assigned to mandatory fields while templates with the **(encvalue,omit)** restriction can also be assigned to optional fields.

If one of the mutation annotation keywords occurs to the right of a *TemplateInstance*, then the *Expression* on the right side of the mutation annotation keyword can use the keyword **value** as an implicit formal parameter to reference the encoded value of that *TemplateInstance*. If the *Expression* does not need to reference the encoded value, then the *TemplateInstance* may be omitted.

If the **@mutation** keyword is used, then the **value** keyword refers to an expression of type **bitstring** and the *Expression* shall evaluate to a value of type **bitstring**.

If the **@mutation_o** keyword is used, then the **value** keyword refers to an expression of type **octetstring** and the *Expression* shall evaluate to a value of type **octetstring**.

If the @**mutation_unichar** keyword is used, then the **value** keyword refers to an expression of type **universal charstring** and the *Expression* shall evaluate to a value of type **universal charstring**. If a different string encoding than the default "UTF-8" is used for the universal charstring, then this string encoding is given as an additional *StringEncoding* operand in parenthesis after the **@mutation_unichar** keyword.

When an encoder processes a value template that is a mutation annotation with a *TemplateInstance*, it will first encode that *TemplateInstance* into a sub-message. It will then transform that sub-message into a TTCN-3 string value of the appropriate type (depending on which mutation annotation is used) and then invoke the evaluation of the mutation *Expression*, using the transformed string value as an actual parameter of the formal parameter **value**. The result of the evaluation is transformed back to a sub-message which is then used instead of the original sub-message as part of the resulting message.

When an encoder processes a value template that is a mutation annotation without a *TemplateInstance*, it will evaluate the mutation *Expression* and transform the resulting value to a sub-message which is then used as the part of the message corresponding to the encoded value.

If the **@mutation_o** keyword is used, the sub-message is transformed into a left-aligned **octetstring** before transformation, so that if the sub-message does not have a bit-length divisible by 8, the appropriate amount of padding bits are the least significant bits of the least significant octet of the **octetstring**. The bit-content of the whole octetstring that is the result of the evaluation will be used as the resulting sub-message.

If the **@mutation_unichar** keyword is used, the sub-message is transformed depending on the given *StringEncoding* into a **universal charstring**. The transformed sub-message should be byte-aligned and have a bit-length that is consistent with the given *StringEncoding* and otherwise an error will be produced. The result of the evaluation is a **universal charstring** that is transformed into a sub-message by using the given *StringEncoding* to encode it into a byte-aligned binary representation.

Restrictions

a) The *Expression* shall conform to the restrictions given in clause 16.4.1 and shall not use any functions with a runs on clause.

b) The *TemplateInstance* shall be an encodable value.

c) The **value** keyword shall not be used inside the *Expression* if no *TemplateInstance* is given.

5.6.2.1 Modifications to ETSI ES 201 873-1 [1], clause 22.2.1 (The Send operation)

The strike-through text shall be replaced by the underlined text as below:

Restrictions

~~a) The TemplateInstance (and all parts of it) shall have a specific value i.e. the use of matching mechanisms such as *AnyValue* is not allowed.~~

a) The *TemplateInstance* shall be an encodable value, i.e. the use of matching mechanisms such as *AnyValue* is not allowed.

5.6.2.2 Modifications to ETSI ES 201 873-1 [1], clause 22.3.1 (The Call operation)

The strike-through text shall be replaced by the underlined text as below:

Restrictions

~~b) All **in** and **inout** parameters of the signature shall have a specific value i.e. the use of matching mechanisms such as *AnyValue* is not allowed.~~

b) All **in** and **inout** parameters of the signature shall be an encodable value, i.e. the use of matching mechanisms such as *AnyValue* is not allowed.

5.6.2.3 Modifications to ETSI ES 201 873-1 [1], clause 22.3.3 (The Reply operation)

The strike-through text shall be replaced by the underlined text as below:

Restrictions

~~c) All **out** and **inout** parameters of the signature shall have a specific value i.e. the use of matching mechanisms such as *AnyValue* is not allowed.~~

c) All **out** and **inout** parameters of the signature shall be encodable values i.e. the use of matching mechanisms such as *AnyValue* is not allowed.

~~g) The *TemplateBody* in the **value** clause shall conform to the template(value) restriction and it shall be type-compatible with the return type specified in the signature of the procedure to which the **reply** operation belongs.~~

g) The *TemplateBody* in the **value** clause shall be an encodable value and it shall be type-compatible with the return type specified in the signature of the procedure to which the **reply** operation belongs.

5.6.2.4 Modifications to ETSI ES 201 873-1 [1], clause 22.3.5 (The Raise operation)

The strike-through text shall be replaced by the underlined text as below:

~~The value part of the **raise** operation consists of the signature reference followed by the exception value.~~

The value part of the **raise** operation consists of the signature reference followed by the exception *TemplateInstance*.

~~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 optional type field in the value specification to the **raise** operation shall be used in cases where it is necessary to avoid any ambiguity of the type of the value being sent.~~

Exception types are specified in the referenced *Signature* declaration. The exception given to the raise operation shall be an encodable value. In cases where it is necessary to avoid any ambiguity of the type of the value being sent, the *TemplateInstance* shall use the inline template syntax with the exception type as prefix.

Restrictions

~~f) The TemplateInstance shall conform to the template(value) restriction (see clause **Error! Reference source not found.**).~~

f) The *TemplateInstance* shall be an encodable value.

5.6.2.5 Modifications to ETSI ES 201 873-1 [1], clause C.5.1 (The encoding function)

The strike-through text shall be replaced by the underlined text as below:

```
encvalue(in template (~~value~~ encvalue) any_type inpar,
         in universal charstring encoding_info := "",
         in universal charstring dynamic_encoding := "") return bitstring
```

The **encvalue** function encodes an encodable value ~~or template~~ into a bitstring. ~~When the actual parameter that is passed to `inpar` is a template, it shall resolve to a specific value (the same restrictions apply as for the argument of the **send** statement).~~ The returned bitstring represents the encoded value of `inpar`, however, the TTCN-3 test system need not make any check on its correctness. The optional `encoding_info` parameter is used for passing additional encoding information to the codec and, if it is omitted, no additional information is sent to the codec.

5.6.2.6 Modifications to ETSI ES 201 873-1 [1], clause C.5.3 (The encoding to universal charstring function)

The strike-through text shall be replaced by the underlined text as below:

```
encvalue_unichar(in template (~~value~~ encvalue) any_type inpar,
                 in charstring string_serialization := "UTF-8",
                 in universal charstring encoding_info := "",
                 in universal charstring dynamic_encoding := "")
                 return universal charstring
```

The **encvalue_unichar** function encodes an encodable value ~~or template~~ into a universal charstring. ~~When the actual parameter that is passed to inpar is a template, it shall resolve to a specific value (the same restrictions apply as for the argument of the **send** statement).~~ The returned universal charstring represents the encoded value of inpar, however, the TTCN-3 test system need not make any check on its correctness. If the optional string_serialization parameter is omitted, the default value "UTF-8" is used. The optional encoding_info parameter is used for passing additional encoding information to the codec and, if it is omitted, no additional information is sent to the codec.

The optional dynamic_encoding parameter is used for dynamic selection of **encode** attribute of the **inpar** ~~value~~ parameter for this single encvalue_unichar call. The rules for dynamic selection of the **encode** attribute are described in clause 27.9.

…

The specific semantics of this function are explained by the following TTCN-3 definition:

```
function encvalue_unichar(in template(~~value~~ encvalue) any_type inpar,
                          in charstring enc
                          in universal charstring encoding_info := "",
                          in universal charstring dynamic_encoding := "")
  return universal charstring {
    return oct2unichar(bit2oct(encvalue(inpar, encoding_info, dynamic_encoding)), enc);
}
```

```
The encvalue_unichar function first invokes the encvalue function in order to encode the
encodable value passed in the inpar parameter to a bitstring. The bitstring is then converted
to an octetstring by the bit2oct function and subsequently to a universal charstring using the
oct2unichar function. The string_serialization parameter defines how the encoded octets (in
fact the encoded bitstring received from the codec) contain the characters. The universal
charstring value is then returned as the result of the encvalue_unichar function.
```

…

5.6.2.7 Modifications to ETSI ES 201 873-1 [1], clause C.5.5 (The encoding to octetstring function)

The strike-through text shall be replaced by the underlined text as below:

```
encvalue_o(in template (~~value~~ encvalue) any_type inpar,
           in universal charstring encoding_info := "",
           in universal charstring dynamic_encoding := "",
           out integer bit_length) return octetstring
```

The **encvalue_o** function encodes an encodable value ~~or template~~ into an octetstring. ~~When the actual parameter that is passed to inpar is a template, it shall resolve to a specific value (the same restrictions apply as for the argument of the **send** statement).~~ The returned octetstring represents the encoded value of inpar, however, the TTCN-3 test system need not make any check on its correctness. In case the encoded message is not octet-based and has a bit length not a multiple of 8, the encoded message will be left-aligned in the returned octetstring and the least significant (8 - (bit length mod 8)) bits in the least significant octet will be 0. The bit length can be assigned to a variable by usage of the formal out parameter bit_length. The optional encoding_info parameter is used for passing additional encoding information to the codec and, if it is omitted, no additional information is sent to the codec.

5.7 Extension of the istemplatekind function

5.7.1 Modifications to ETSI ES 201 873-1 [1], clause C.3.5 (Matching mechanism detection)

Extend the first clause as follows:

This function allows to examine if a template contains a certain kind of the matching mechanisms or mutation.

Add the following new paragraph 5:

If the enquired kind of template is a mutation (clause 5.6 of ETSI ES 203 022), the function shall return **`true`** if the template in the invalue parameter contains a mutation on the first level of nesting.

Extend table C.1 (Allowed values of kind parameter) with the following new rows:

Value of kind parameter	Searched matching mechanism	
	Name	Clause reference
"dynamic"	Dynamic matching	5.1 of ETSI ES 203 022
"conjunction"	Conjunction	5.3.1 of ETSI ES 203 022
"implication"	Implication	5.3.2 of ETSI ES 203 022
"exclusion"	Exclusion	5.3.3 of ETSI ES 203 022
"disjunction"	Disjunction	5.3.4 of ETSI ES 203 022
"repetition"	Repetition	5.4 of ETSI ES 203 022
"mutation"	Mutation	5.6 of ETSI ES 203 022
"concatenation"	Concatenation	5.4 of ETSI ES 203 022

5.7.2 Modifications to ETSI ES 201 873-1 [1], clause E.2.2.5 (Matching mechanism detection)

Extend the TemplateKind type definition as follows:

```
type charstring TemplateKind ("value", "list", "complement", "AnyValue", "?", "AnyValueOrNone",
"*", "range", "subset", "superset", "omit", "decmatch", "AnyElement", "AnyElementsOrNone",
"permutation", "length", "ifpresent", "pattern", "dynamic", "conjunction", "implication",
"exclusion", "disjunction", "repetition", "mutation", "concatenation");
```

NOTE: This definition is compatible with version 4.11.1 of ETSI ES 201 873-1 [1]. If a later version change the definition of the TemplateKind type, the value list constraints added by this document, shown above in underline text, should be added to the changed definition to be compatible with that version of the TTCN-3 core language standard.

Add the following useful constant definitions:

```
const TemplateKind DYNAMIC := "dynamic";
const TemplateKind CONJUNCTION := "conjuction";
const TemplateKind IMPLICATION := "implication";
const TemplateKind EXCLUSION := "exclustion";
const TemplateKind DISJUNCTION := "disjunction";
const TemplateKind REPETITION := "repetition";
const TemplateKind MUTATION := "mutation";
const TemplateKind CONCATENATION := "concatenation";
```

6 TCI Extensions for the Package

6.1 Extensions to clause 7.2.2.2.0 of ETSI ES 201 873-6 [4], Basic rules

Figure 4 of ETSI ES 201 873-6 [4] shall be extended as shown on the below figure 6.1 of the present document.

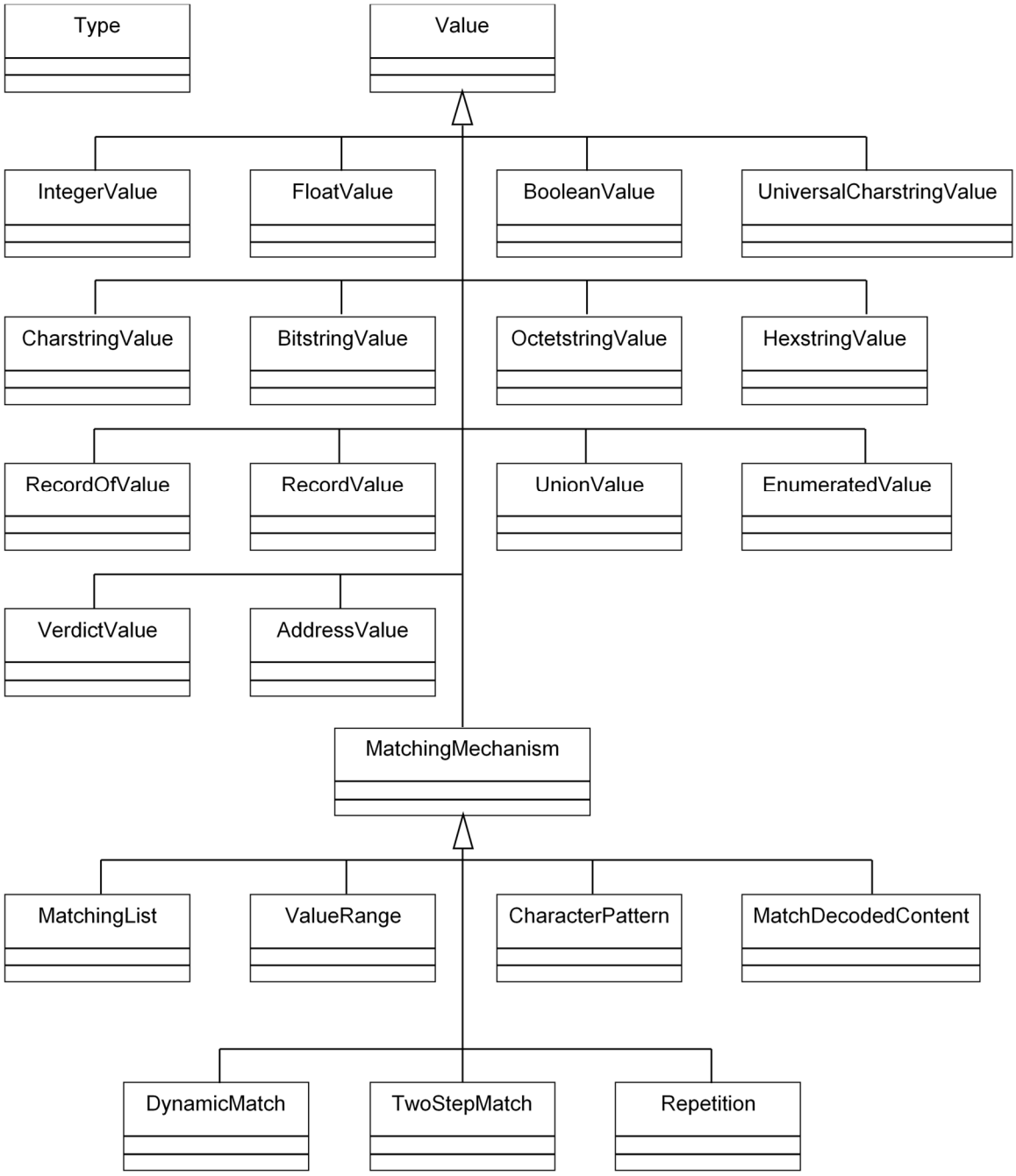

Figure 6.1: Extended hierarchy of abstract values

6.2 Extensions to clause 7.2.2.3.1 of ETSI ES 201 873-6 [4], The abstract data type MatchingMechanism

The definition of `TciMatchingTypeType` is to be extended with constants for the matching mechanisms defined in the present document:

`TciMatchingTypeType getMatchingType()`
: Returns the matching mechanism type. A value of `TciMatchingTypeType` can have one of the following constants: `TEMPLATE_LIST, COMPLEMENTED_LIST, ANY_VALUE, ANY_VALUE_OR_NONE, VALUE_RANGE, SUBSET, SUPERSET, ANY_ELEMENT, ANY_ELEMENTS_OR_NONE, PATTERN, DECODED_MATCH, DYNAMIC_MATCHING, CONJUNCTION, IMPLICATION, EXCLUSION, DISJUNCTION, REPETITION.`

6.3 Extensions to clause 7.2.2.3.2 of ETSI ES 201 873-6 [4], The abstract data type MatchingList

Clause 7.2.2.3.2 The abstract data type MatchingList

The first paragraph of clause 7.2.2.3.2 is to be extended by adding conjunction and disjunction to the list of allowed matching mechanisms:

The abstract data type `MatchingList` is used to represent matching mechanisms that contain a list of items of the same type: template list, complemented template list, SubSet, SuperSet, conjunction and disjunction.

6.4 Extensions to clause 7.2.2.3 of ETSI ES 201 873-6 [4], The abstract data type MatchingMechanism

In order to support the dynamic match, implication, exclusion and repetition matching mechanism, the `DynamicMatch`, `TwoStepMatch` and `Repetition` abstract data types are added to the clause 7.2.2.3 of ETSI ES 201 873-6 [4].

Clause 7.2.2.3.6 The abstract data type DynamicMatch

This clause is to be added.

The abstract data type `DynamicMatch` is used to represent the dynamic matching mechanism. The TCI support of this matching mechanism is limited, e.g. it is only possible to create dynamic matching based on functions.

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

`TBoolean isFunctionBased ()`
: Returns `true` if the mechanism uses short-hand notation **`@dynamic`** *FunctionRef* and `false` otherwise.

`TciBehaviourIdType getMatchingFunction ()`
: Returns the qualified name of the associated function if the mechanism uses the short-hand notation **`@dynamic`** *FunctionRef*. The distinct value `null` is returned otherwise.

`void setMatchingFunction (TciBehaviourIdType functionId)`
: Sets the function associated with the matching mechanism. After calling the function, the matching mechanism will behave as a short-hand notation **`@dynamic`** `fuctionId`. The function referenced in the functionId parameter shall fulfil all requirements described in the clause 4.1.

Clause 7.2.2.3.7 The abstract data type TwoStepMatch

This clause is to be added.

The abstract data type `TwoStepMatch` is used to represent the implication and exclusion matching mechanisms. The data type can access both templates that form these matching mechanism. The first template (i.e. precondition of the implication matching mechanism and general template of the exclusion matching mechanism) is called *primary template* and the second template (i.e. the implied template of the implication matching mechanism and excluded template of the exclusion matching mechanism) is called *secondary template*.

When working with instances received from the `getPrimaryMatch` and `getSecondaryMatch` methods and passed to the `setPrimaryMatch` and `setSecondaryMatch` method, no assumption shall be made on how the data are stored in a matching mechanism. An internal implementation might choose to use a reference to the data or to copy the data. It is safe to assume that the data are copied. Therefore, it should be assumed that subsequent modifications of the data will not be considered in the `TwoStepMatch` instance.

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

`Value getPrimaryMatch ()` Returns the primary template.

`void setPrimaryMatch (Value content)`
Sets the primary template.

`Value getSecondaryMatch ()` Returns the secondary template.

`void setSecondaryMatch (Value content)`
Sets the secondary template.

Clause 7.2.2.3.8 The abstract data type Repetition

This clause is to be added.

The abstract data type `Repetition` is used to represent the repetition matching mechanism.

When working with instances received from the `getRepeatedTemplate` method and passed to the `setRepeatedTemplate` method, no assumption shall be made on how the data are stored in a matching mechanism. An internal implementation might choose to use a reference to the data or to copy the data. It is safe to assume that the data are copied. Therefore, it should be assumed that subsequent modifications of the data will not be considered in the `Repetition` instance.

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

`Value getRepeatedTemplate ()` Returns the repeated template.

`void setRepeatedTemplate (Value repeatedTemplate)`
Sets the repeated template.

`LengthRestriction getRepetitionCount ()`
Returns how many times the template shall match.

`void setRepetitionCount (LengthRestriction repetition)`
Sets how many times the template shall match.

6.5 Extensions to clause 8 of ETSI ES 201 873-6 [4], Java™ language mapping

Clause 8.3.2.17 TciMatchingTypeType

This clause is to be extended.

```
// TCI IDL TciMatchingTypeType
package org.etsi.ttcn.tci;
public interface TciMatchingType {
    public final static int TEMPLATE_LIST          = 0 ;
    public final static int COMPLEMENTED_LIST      = 1 ;
    public final static int ANY_VALUE              = 2 ;
```

```
    public final static int ANY_VALUE_OR_NONE       = 3 ;
    public final static int VALUE_RANGE             = 4 ;
    public final static int SUBSET                  = 5 ;
    public final static int SUPERSET                = 6 ;
    public final static int ANY_ELEMENT             = 7 ;
    public final static int ANY_ELEMENTS_OR_NONE    = 8 ;
    public final static int PATTERN                 = 9 ;
    public final static int MATCH_DECODED_CONTENT   = 10 ;
    public final static int OMIT_TEMPLATE           = 11 ;
    public final static int DYNAMIC_MATCHING        = 12 ;
    public final static int CONJUNCTION             = 13 ;
    public final static int IMPLICATION             = 14 ;
    public final static int EXCLUSION               = 15 ;
    public final static int DISJUNCTION             = 16 ;
    public final static int REPETITION              = 17 ;
}
```

Clause 8.3.5.6 DynamicMatch

This clause is to be added.

`DynamicMatch` is mapped to the following interface:

```
// TCI IDL DynamicMatch
package org.etsi.ttcn.tci;
public interface DynamicMatch {
    public Boolean isFunctionBased ();
    public TciBehaviourId getMatchingFunction ();
    public void setMatchingFunction (TciBehaviourId functionId);
}
```

Methods:

- `isFunctionBased` Returns `true` if the mechanism uses the short-hand notation **`@dynamic`** *FunctionRef* and `false` otherwise.
- `getMatchingFunction` Returns the qualified name of the associated function.
- `setMatchingFunction` Sets the function associated with the matching mechanism.

Clause 8.3.5.7 TwoStepMatch

This clause is to be added.

`TwoStepMatch` is mapped to the following interface:

```
// TCI IDL TwoStepMatch
package org.etsi.ttcn.tci;
public interface TwoStepMatch {
    public Value getPrimaryTemplate ();
    public void setPrimaryTemplate (Value primaryTemplate);
    public Value getSecondaryTemplate ();
    public void setSecondaryTemplate (Value secondaryTemplate);
}
```

Methods:

- `getPrimaryTemplate` Returns the primary template.
- `setPrimaryTemplate` Sets the primary template.
- `getSecondaryTemplate` Returns the secondary template.
- `setSecondaryTemplate` Sets the secondary template.

Clause 8.3.5.8 Repetition

This clause is to be added.

`Repetition` is mapped to the following interface:

```
// TCI IDL Repetition
package org.etsi.ttcn.tci;
public interface Repetition {
    public Value getRepeatedTemplate ();
    public void setRepeatedTemplate (Value primaryTemplate);
    public LengthRestriction getRepetitionCount ();
    public void setRepetitionCount (LengthRestriction repetitionCount);
}
```

Methods:

- `getRepeatedTemplate` Returns the repeated template.
- `setRepeatedTemplate` Sets the repeated template.
- `getRepetitionCount` Returns the repetition count.
- `setRepetitionCount` Sets the repetition count.

6.6 Extensions to clause 9 of ETSI ES 201 873-6 [4], ANSI C language mapping

Clause 9.2 Data

Table 5 is to be extended.

TCI IDL Interface	ANSI C representation	Notes and comments
:		
DynamicMatch		
TBoolean isFunctionBased ()	`Boolean tciIsMatchFunctionBased (Value inst)`	
QualifiedName getMatchingFunction()	`QualifiedName * tciGetMatchingFunction (Value inst)`	
Void setMatchingFunction (QualifiedName functionId)	`void tciSetMatchingFunction(Value inst, QualifiedName functionId)`	
TwoStepMatch		
Value getPrimaryTemplate()	`Value getPrimaryTamplate(Value inst)`	
void setPrimaryTemplate(Value template)	`void setPrimaryTemplate(Value inst, Value template)`	
Value getSecondaryTemplate()	`Value getSecondaryTamplate(Value inst)`	
void setSecondaryTemplate(Value template)	`void setSecondaryTemplate(Value inst, Value template)`	
Repetition		
Value getRepeatedTemplate()	`Value getRepeatedTemplate(Value inst)`	
void setRepeatedTemplate(Value template)	`void setRepeatedTemplate(Value inst, Value template)`	
LengthRestriction getRepeatedTemplate()	`TciLengthRestriction getRepetitionCount(Value inst)`	
Void setRepetitionCount (LengthRestriction repetitionCount)	`void setRepetitionCount(Value inst, TciLengthRestriction repetitionCount)`	

Clause 9.5 Data

Table 7 is to be extended.

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

TciMatchingTypeType	typedef enum
	{
	  TCI_TEMPLATE_LIST = 0,
	  TCI_COMPLEMENTED_LIST = 1,
	  TCI_ANY_VALUE = 2,
	  TCI_ANY_VALUE_OR_NONE = 3,
	  TCI_VALUE_RANGE = 4,
	  TCI_SUBSET = 5,
	  TCI_SUPERSET = 6,
	  TCI_ANY_ELEMENT = 7,
	  TCI_ANY_ELEMENTS_OR_NONE = 8,
	  TCI_PATTERN = 9,
	  TCI_MATCH_DECODED_CONTENT = 10,
	  TCI_OMIT_TEMPLATE = 11,
	  TCI_DYNAMIC_MATCHING = 12,
	  TCI_CONJUNCTION = 13,
	  TCI_IMPLICATION = 14,
	  TCI_EXCLUSION = 15,
	  TCI_DISJUNCTION = 16,
	  TCI_REPETITION = 17
	} TciMatchingTypeType;	

:		

6.7 Extensions to clause 10 of ETSI ES 201 873-6 [4], C++ language mapping

Clause 10.5.2.16 TciMatchingTypeType

This clause is to be extended.

```
typedef enum
{
  TCI_TEMPLATE_LIST = 0,
  TCI_COMPLEMENTED_LIST = 1,
  TCI_ANY_VALUE = 2,
  TCI_ANY_VALUE_OR_NONE = 3,
  TCI_VALUE_RANGE = 4,
  TCI_SUBSET = 5,
  TCI_SUPERSET = 6,
  TCI_ANY_ELEMENT = 7,
  TCI_ANY_ELEMENTS_OR_NONE = 8,
  TCI_PATTERN = 9,
  TCI_MATCH_DECODED_CONTENT = 10,
  TCI_OMIT_TEMPLATE = 11,
  TCI_DYNAMIC_MATCHING = 12,
  TCI_CONJUNCTION = 13,
  TCI_IMPLICATION = 14,
  TCI_EXCLUSTION = 15,
  TCI_DISJUNCTION = 16,
  TCI_REPETITION = 17
} TciMatchingType;
```

Clause 10.5.3.23 DynamicMatch

This clause is to be added.

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

```
class DynamicMatch : public virtual MatchingMechanism {
public:
    virtual ~DynamicMatch ();
    virtual Tboolean isFunctionBased () const =0;
    virtual const TciBehaviourId * getMatchingFunction () const =0;
    virtual void setMatchingFunction (const TciBehaviourId & functionId) =0;
    virtual Tboolean operator== (const DynamicMatch &p_dynamicMatch) const =0;
    virtual DynamicMatch * clone () const =0;
```

```
    virtual Tboolean operator< (const DynamicMatch &p_content) const =0;
}
```

Methods:

`~DynamicMatch`
: Destructor

`isFunctionBased`
: Returns `true` if the mechanism uses the short-hand notation **`@dynamic`** *FunctionRef* and `false` otherwise

`getMatchingFunction`
: Returns the qualified name of the associated function

`setMatchingFunction`
: Sets the function associated with the matching mechanism

`operator==`
: Returns true if both objects are equal

`clone`
: Return a copy of the matching mechanism

`operator<`
: Operator < overload

Clause 10.5.3.24 TwoStepMatch

This clause is to be added.

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

```
class TwoStepMatch : public virtual MatchingMechanism {
public:
    virtual ~TwoStepMatch ();
    virtual Value & getPrimaryTemplate () const =0;
    virtual void setPrimaryTemplate (const Value & template) =0;
    virtual Value & getSecondaryTemplate () const =0;
    virtual void setSecondaryTemplate (const Value & template) =0;
    virtual Tboolean operator== (const TwoStepMatch &p_twoStepMatch) const =0;
    virtual TwoStepMatch * clone () const =0;
    virtual Tboolean operator< (const TwoStepMatch &p_content) const =0;
}
```

Methods:

`~TwoStepMatch`
: Destructor

`getPrimaryTemplate`
: Returns the primary template

`setPrimaryTemplate`
: Sets the primary template

`getSecondaryTemplate`
: Returns the secondary template

`setSecondaryTemplate`
: Sets the secondary template

`operator==`
: Returns true if both objects are equal

`clone`
: Return a copy of the matching mechanism

`operator<`
: Operator < overload

Clause 10.5.3.24 Repetition

This clause is to be added.

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

```
class Repetition : public virtual MatchingMechanism {
public:
    virtual ~Repetition ();
    virtual Value & getRepeatedemplate () const =0;
    virtual void setRepeatedTemplate (const Value & template) =0;
    virtual LengthRestriction & getRepetitionCount () const =0;
    virtual void setRepetitionCount (const LengthRestriction & repetitionCount) =0;
    virtual Tboolean operator== (const Repetition &p_repetition) const =0;
    virtual Repetition * clone () const =0;
```

```
    virtual Tboolean operator< (const Repetition &p_content) const =0;
}
```

Methods:

`~Repetition`
: Destructor

`getRepeatedTemplate`
: Returns the repeated template

`setRepeatedTemplate`
: Sets the repeated template

`getRepetitionCount`
: Returns repetition count

`setRepetitionCount`
: Sets repetition count

`operator==`
: Returns true if both objects are equal

`clone`
: Return a copy of the matching mechanism

`operator<`
: Operator < overload

6.8 Extensions to clause 12 of ETSI ES 201 873-6 [4], C# language mapping

Clause 12.4.2.16 TciMatchingTypeType

This clause is to be extended.

```
public enum TciMatchingType {
    TemplateList = 0,
    ComplementedList = 1,
    AnyValue = 2,
    AnyValueOrNone = 3,
    ValueRange = 4,
    Subset = 5,
    Superset = 6,
    AnyElement = 7,
    AnyElementsOrNone = 8,
    Pattern = 9,
    MatchDecodedContent = 10,
    OmitTemplate = 11,
    DynamicMatch = 12,
    Conjunction = 13,
    Implication = 14,
    Exclusion = 15,
    Disjunction = 16,
    Repetition = 17
}
```

Clause 12.4.5.6 DynamicMatch

This clause is to be added.

The IDL type **`DynamicMatch`** is mapped to the following interface:

```
public interface ITciDynamicMatch : IMatchingMechanism
{
    bool IsFunctionBased { get; }
    ITciBehaviourId MatchingFunction { get; set; }
}
```

Methods:

- `IsFunctionBased` Returns `true` if the mechanism uses short-hand notation **`@dynamic`** *FunctionRef* and `false` otherwise.
- `MatchingFunction` Gets or sets the function associated with the matching mechanism.

Clause 12.4.5.7 TwoStepMatch

This clause is to be added.

The IDL type **`TwoStepMatch`** is mapped to the following interface:

```
public interface ITciTwoStepMatch : IMatchingMechanism
{
    ITciValue PrimaryTemplate { get; set; }
    ITciValue SecondaryTemplate { get; set; }
}
```

Methods:

- `PrimaryTemplate` Gets or sets the primary template.
- `SecondaryTemplate` Gets or sets the secondary template.

Clause 12.4.5.8 Repetition

This clause is to be added.

The IDL type **`Repetition`** is mapped to the following interface:

```
public interface ITciRepetition : IMatchingMechanism
{
    ITciValue RepeatedTemplate { get; set; }
    ITciLengthRestriction RepetitionCount { get; set; }
}
```

Methods:

- `RepeatedTemplate` Gets or sets the repeated template.
- `RepetitionCount` Gets or sets the repetition count

Annex A (normative): BNF and static semantics

A.0 Additional TTCN-3 terminals

TTCN-3 special terminal symbols defined in this package are listed in table A.1.

Table A.1: List of TTCN-3 special terminal symbols

Repetition mark	#

Table A.2 presents all additional TTCN-3 terminals which are reserved words when using this package. Like the reserved words defined in the TTCN-3 core language, the TTCN-3 terminals listed in table A.2 shall not be used as identifiers in a TTCN-3 module. These terminals shall be written in all lowercase letters.

Table A.2: List of additional TTCN-3 terminals which are reserved words

conjunct	**disjunct**	**implies**	

TTCN-3 modifiers defined in this package are listed in table A.3.

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

@dynamic	**@match**		

A.1 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 [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.

```
92. TemplateBody ::= (SimpleSpec |
                  FieldSpecList |
                  ArrayValueOrAttrib
                  ) [ExtraMatchingAttributes] [ValueRedirect]

107. MatchingSymbol ::= Complement |
                        (AnyValue [WildcardLengthMatch]) |
                        (AnyOrOmit [WildcardLengthMatch]) |
                        ListOfTemplates |
                        Range |
                        BitStringMatch |
                        HexStringMatch |
                        OctetStringMatch |
                        CharStringMatch |
                        SubsetMatch |
                        SupersetMatch |
                        DecodedContentMatch |
                        DynamicMatch |
                        ConjunctionMatch |
                        ImplicationMatch |
                        ExclusionMatch
106. ArrayElementSpec ::= Minus |
                        PermutationMatch |
                        TemplateBody |
                        DisjunctionMatch |
                        RepetitionMatch
112. BinOrMatch ::= Bin [StringRepetition] |
                    AnyValue [StringRepetition] |
                    AnyOrOmit
114. HexOrMatch ::= Hex [StringRepetition] |
```

```
                    AnyValue [StringRepetition] |
                    AnyOrOmit
116. OctOrMatch ::= Oct [StringRepetition] |
                    AnyValue [StringRepetition] |
                    AnyOrOmit
250.VarInstance ::= VarKeyword (
                   ([LazyModifier | FuzzyModifier] Type VarList) |
                   ((TemplateKeyword | RestrictedTemplate)
                    [LazyModifier | FuzzyModifier]
                    [MatchModifier]
                    Type TempVarList)
                   )
413. Value ::= PredefinedValue | ReferencedValue | SpecialValue
414. PredefinedValue ::= Bstring |
                         BooleanValue |
                         CharStringValue |
                         Number | /* IntegerValue */
                         Ostring |
                         Hstring |
                         VerdictTypeValue |
                         Identifier | /* EnumeratedValue */
                         FloatValue
457.FormalTemplatePar ::= [(InParKeyword | OutParKeyword | InOutParKeyword )]
                           (TemplateKeyword | RestrictedTemplate)
                           [(LazyModifier | FuzzyModifier)]
                           [MatchModifier]
                           Type Identifier [ArrayDef] [":=" (TemplateInstance | Minus)]
```

A.2 Deleted TTCN-3 syntax BNF productions

The rule for the nonterminal AddressValue shall be deleted.

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 "0222".

```
022001. DynamicMatch ::= DynamicModifier (StatementBlock | FunctionRef)
022002. DynamicModifier::= "@dynamic"

022003. RepetitionMatch ::= TemplateBody RepetitionCountSpec
022004. RepetitionCountSpec ::= "#" "(" [SingleExpression] ["," [SingleExpression]] ")"

/** STATIC SEMANTICS The ValueKeyword has only meaning as a value in the StatementBlock of a 022001.
DynamicMatch. The IndexModifier has only meaning as a value in the TemplateBody of a
RepetitionMatch. **/
022005. SpecialValue ::= ValueKeyword | NullValue | IndexModifier | OmitKeyword
022006. NullValue ::= "null"

022007. ConjunctionMatch ::= ConjunctKeyword ListOfTemplates
022008. ConjunctKeyword ::= "conjunct"
022009. ImplicationMatch ::= TemplateInstance ImpliesKeyword InfixTemplateOperand
022010. ImpliesKeyword ::= "implies"
022011. InfixTemplateOperand ::= ( TemplateInstanceNoAttr | ( "(" TemplateInstance ")" ))
022012. TemplateInstanceNoAttr ::= [(Type | Signature) Colon] [DerivedRefWithParList AssignmentChar]
                                  TemplateBodyNoAttr
022013. TemplateBodyNoAttr ::= SimpleSpec |
                               FieldSpecList |
                               ArrayValueOrAttrib
022014. ExclusionMatch ::= TemplateInstance ExceptKeyword InfixTemplateOperand
022015. DisjunctionMatch ::= DisjunctKeyword ListOfTemplates
022016. DisjunctKeyword ::= "disjunct"
022017. ValueRedirect ::= "->" VariableRef
022018. StringRepetition ::= "#" (Num | ( "(" [Number] ["," ] [Number] ")" ))
022019. MatchModifier ::= "@match"
```

History

Document history		
V1.1.1	July 2017	Publication
V1.2.1	May 2018	Publication
V1.3.1	February 2019	Membership Approval Procedure MV 20190406: 2019-02-05 to 2019-04-08