Final draft ETSI ES 201 873-9 V4.9.1 (2018-03)

Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 9: Using XML schema with TTCN-3

Reference
RES/MTS-201873-9 T3ed491

Keywords
language, testing, TTCN-3, XML

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 is part 9 of a multi-part deliverable. Full details of the entire series can be found in part 1 [1].

Modal verbs terminology

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

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

1 Scope

The present document defines the mapping rules for W3C® XML Schema (as defined in [7] to [9]) to TTCN-3 as defined in ETSI ES 201 873-1 [1] to enable testing of XML-based systems, interfaces and protocols.

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] Void.

[3] Recommendation ITU-T X.680: "Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation".

[4] Recommendation ITU-T X.694: "Information technology - ASN.1 encoding rules: Mapping W3C XML schema definitions into ASN.1".

[5] World Wide Web Consortium W3C Recommendation: "Extensible Markup Language (XML) 1.1".

NOTE: Available at http://www.w3.org/TR/xml11.

[6] World Wide Web Consortium W3C Recommendation (2006): "Namespaces in XML 1.0".

NOTE: Available at http://www.w3.org/TR/REC-xml-names/.

[7] World Wide Web Consortium W3C Recommendation (2004): "XML Schema Part 0: Primer".

NOTE: Available at http://www.w3.org/TR/xmlschema-0.

[8] World Wide Web Consortium W3C Recommendation (2004): "XML Schema Part 1: Structures".

NOTE: Available at http://www.w3.org/TR/xmlschema-1.

[9] World Wide Web Consortium W3C Recommendation (2004): "XML Schema Part 2: Datatypes".

NOTE: Available at http://www.w3.org/TR/xmlschema-2.

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] World Wide Web Consortium W3C Recommendation: "SOAP version 1.2, Part 1: Messaging Framework".

NOTE: Available at http://www.w3.org/TR/soap12.

[i.2] ISO 8601 (2004): "Data elements and interchange formats - Information interchange - Representation of dates and times".

[i.3] Void.

[i.4] ISO/IEC 10646 (2012): "Information technology - Universal Coded Character Set (UCS)".

[i.5] ETSI ES 202 781: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Configuration and Deployment Support".

[i.6] Void.

[i.7] Void.

[i.8] Void.

[i.9] Void.

[i.10] Void.

[i.11] 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.12] 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.13] ETSI ES 201 873-11: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 11: Using JSON with TTCN-3".

[i.14] W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes.

NOTE: Available at http://www.w3.org/TR/xmlschema11-2.

3 Definitions and abbreviations

3.1 Definitions

For the purposes of the present document, the terms and definitions given in ETSI ES 201 873-1 [1], Recommendation ITU-T X.694 [4] and the following apply:

alphabetical order: way of sorting the XSD names based on the code positions of their characters according to ISO/IEC 10646 [i.4]

NOTE: During this sorting the group, plane, row and cell octets is considered, in this order. Names, starting with a character with a smaller code position take precedence Among the names with identical first character, names containing no more characters take precedence over all other names. Otherwise, names with the second character of smaller code position take precedence, etc. This algorithm is to be continued recursively until all names are sorted into a sequential order.

schema component: generic XSD term for the building blocks that comprise the abstract data model of the schema

NOTE: The primary components, which may (type definitions) or obliged to (element and attribute declarations) have names are as follows: simple type definitions, complex type definitions, attribute declarations and element declarations. The secondary components, which are obliged to have names, are as follows: attribute group definitions, identity-constraint definitions, model group definitions and notation declarations. Finally, the "helper" components provide small parts of other components; they are not independent of their context: annotations, model groups, particles, wildcards and attribute uses.

schema document: XML document containing a collection of schema components, assembled in a *schema* element information item

NOTE: The target namespace of the schema document may be defined (specified by the *targetNamespace* attribute of the *schema* element) or may be absent (identified by a missing *targetNamespace* attribute of the *schema* element). The latter case is handled in the present document as a particular case of the target namespace being defined.

target TTCN-3 module: TTCN-3 module, generated during the conversion, to which the TTCN-3 definition produced by the translation of a given XSD declaration or definition is added

XML Schema: set of schema documents forming a complete specification (i.e. all definitions and references are completely defined)

NOTE: The set may be composed of one or more schema documents, and in the latter case identifying one or more target namespaces (including absence of the target namespace) and more than one schema documents of the set may have the same target namespace (including absence of the target namespace).

xsi: attributes: XML attribute stipulating the content of schema-instances (schema-valid XML documents)

NOTE 1: XSD defines several attributes for direct use in any XML documents.

NOTE 2: These attributes are in the namespace http://www.w3.org/2001/XMLSchema-instance. By convention these XML attributes are referred to by using the prefix `"xsi: "`, though in practice, any prefix can be used.

3.2 Abbreviations

For the purposes of the present document, the following abbreviations apply:

ASN.1	Abstract Syntax Notation One
DTD	Document Type Description
LF	Line Feed
SOAP	Simple Object Access Protocol
SUT	System Under Test
TTCN-3	Testing and Test Control Notation version 3

URI	Uniform Resource Identifier
UTF-8	Unicode Transformation Format-8
W3C®	World Wide Web Consortium
XML	eXtensible Markup Language
XSD	XML Schema Definition

4 Introduction

An increasing number of distributed applications use the XML format to exchange data for various purposes like data bases queries or updates or event telecommunications operations such as provisioning. All of these data exchanges follow very precise rules for data format description in the form of Document Type Description (DTD) [5] and [6] or more recently the proposed XML Schemas [7], [5] and [9]. There are even some XML based communication protocols like SOAP [i.1] that are based on XML Schemas. Like any other communication-based systems, components and protocols, XML based systems, components and protocols are candidates for testing using TTCN-3 [1]. Consequently, there is a need for establishing a mapping between XML data description techniques like DTD or Schemas to TTCN-3 standard data types.

The core language of TTCN-3 is defined in ETSI ES 201 873-1 [1] and provides a full text-based syntax, static semantics and operational semantics as well as a definition for the use of the language with ASN.1 in ETSI ES 201 873-7 [i.11]. The XML mapping provides a definition for the use of the core language with XML Schema structures and types, enabling integration of XML data with the language as shown in figure 1.

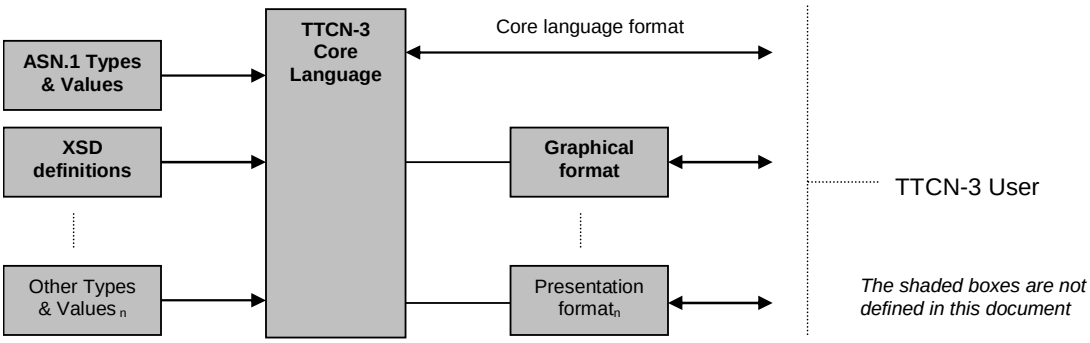

Figure 1: User's view of the core language and the various presentation formats

For compatibility reasons, it was the purpose of the present document that the TTCN-3 code obtained from the XML Schema using the explicit mapping will be the same as the TTCN-3 code obtained from first converting the XML Schema using Recommendation ITU-T X.694 [4] into ASN.1 [3] and then converting the resulting ASN.1 code into TTCN-3 according to ETSI ES 201 873-7 [i.11]. However, due to the specifics of testing, in a few cases the present document will produce a superset or different constructs of what Recommendation ITU-T X.694 [4] would produce. For example, according to Recommendation ITU-T X.694 [4], abstract elements are omitted when converting the head element of a substitution group, while the present document includes also the abstract elements into the resulted **`union`** type, thus allowing provoking the SUT with incorrect data.

5 Mapping XML Schemas

5.0 Approach

There are two approaches to the integration of XML Schema and TTCN-3, which will be referred to as implicit and explicit mapping. The implicit mapping makes use of the import mechanism of TTCN-3, denoted by the keywords *language* and *import*. It facilitates the immediate use of data specified in other languages. Therefore, the definition of a specific data interface for each of these languages is required. The explicit mapping translates XML Schema definitions directly into appropriate TTCN-3 language artefacts.

In case of an implicit mapping an internal representation shall be produced from the XML Schema, which representation shall retain all the structural and encoding information. This internal representation is typically not accessible by the user. To make the internal representations related to a given target namespace referenceable in a TTCN-3 module, the module shall explicitly import the target namespace, using its TTCN-3 name equivalent resulting from applying clause 5.2.2 to the namespace. The TTCN-3 import statement shall use the language identifier string specified below. TTCN-3 data types described in clause 6 (equivalents to built-in XSD types), in case of an implicit conversion, are internal to the tool and can be referenced in TTCN-3 modules importing any target namespaces of an XSD document explicitly. These types can be also referenced in TTCN-3 modules that explicitly import the XSD module (see annex A). In this case, the import clause refers to the tool's internal representation of the XSD data types and not to an existing module. When importing from an XSD Schema using implicit mapping, the following language identifier string shall be used:

- "XSD".

For explicit mapping, the information present in the XML Schema shall be mapped into accessible TTCN-3 code and - the XML structural information which does not have its correspondent in TTCN-3 code - into accessible encoding instructions. In case of an explicit conversion the TTCN-3 data types described in clause 6 (equivalents to built-in XSD types) are not visible in TTCN-3 by default, the user shall import the XSD module (see annex A) explicitly, in addition to the TTCN-3 modules resulted from the conversion. When importing TTCN-3 modules generated by explicit conversion, the use of the "XSD" language clause is optional, but if used, the imported TTCN-3 module shall be appended with one of the XML encode attributes, specified in clause B.2.

The mapping shall start on a set of valid XSD *schema*-s and shall result in a set of valid TTCN-3 modules.

All XSD definitions are **`public`** by default (see clause 8.2.3 of ETSI ES 201 873-1 [1]).

The examples of the present document are written in the assumption of explicit mapping, although the difference is mainly in accessibility and visibility of generated TTCN-3 code and encoding instruction set.

The present document is structured in three distinct parts:

- Clause 6 "Built-in data types" defines the TTCN-3 mapping for all basic XSD data types like strings (see clause 6.2), integers (see clause 6.3), floats (see clause 6.4), etc. and facets (see clause 6.1) that allow for a simple modification of types by restriction of their properties (e.g. restricting the length of a string or the range of an integer).

- Clause 7 "Mapping XSD components" covers the translation of more complex structures that are formed using the components shown in table 1 and a set of XSD attributes (see clause 7.1) which allow for modification of constraints of the resulting types.

- Clause 8 "Substitution" covers the translation of more XSD elements and types that may be substituted for other XSD elements or types respectively in instance documents.

Table 1: Overview of XSD constructs

Element	Defines tags that can appear in a conforming XML document.
attribute	Defines attributes for element tags in a conforming XML document.
simpleType	Defines the simplest types. They may be a built-in type, a list or choice of built-in types and they are not allowed to have attributes.
complexType	Defines types that are allowed to be composed, e.g. have attributes and an internal structure.
named model group	Defines a named group of elements.
attribute group	Defines a group of attributes that can be used as a whole in definitions of complexTypes.
identity constraint	Defines that a component has to exhibit certain properties in regard to uniqueness and referencing.

5.1 Namespaces and document references

5.1.1 Namespaces

A single XML Schema may be composed of a single or several *schema* element information items, and shall be translated to one or more TTCN-3 modules, corresponding to *schema* components that have the same target namespace, including no target namespace. For XSD *schema*s with the same target namespace (including absence of the target namespace) exactly one TTCN-3 module shall be generated.

The names of the TTCN-3 modules generated based on this clause shall be the result of applying the name transformation rules in clause 5.2.2 to the related target namespace, if it exists, or to the predefined name "NoTargetNamespace".

NOTE 1: More than one *schema* element information items in an XML Schema may have the same target namespace, including the case of no target namespace.

The information about the target namespaces and prefixes from the *targetNamespace* and *xmlns* attributes of the corresponding *schema* elements, if exist, shall be preserved in the encoding instruction "namespace as…" attached to the TTCN-3 module. If the target namespace is absent, no "namespace as …" encoding instruction shall be attached to the TTCN-3 module. All declarations in the module shall inherit the target namespace of the module (including absence of the target namespace).

NOTE 2: If different *schema* element information items using the same target namespace associates different prefixes to that namespace, it is a tool implementation option, which prefix is preserved in the "namespace as…" encoding instruction.

EXAMPLE: Schemas with the same namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:ns1="http://www.example.org"
            targetNamespace="http://www.example.org">
    <!-- makes no difference if this schema is including the next one -->
:
</xsd:schema>

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:ns2="http://www.example.org"
            targetNamespace="http://www.example.org">
    <!-- makes no difference if this schema is including the previous one -->
:
</xsd:schema>
```

Will result e.g. in the following TTCN-3 module:

```
module http_www_example_org {
: // the content of the module is coming from both schemas
}
with {
    encode "XML";
    variant "namespace as 'http://www.example.org' prefix 'ns1'";
```

```
// the prefix in the encoding instruction could also be 'ns2', this is a tool's option.
}
```

5.1.2 Includes

XSD *include* element information items shall be ignored if the included *schema* element has the same target namespace as the including one (implying the absence of the target namespace). If the included *schema* element has no target namespace but the including *schema* has (i.e. it is not absent), all definitions of the included *schema* shall be mapped twice, i.e. the resulted TTCN-3 definitions shall be inserted to the TTCN-3 module generated for the *schema* element(s) with no target namespace as well as to the module generated for the *schema* element(s) with the target namespace of the including *schema*.

EXAMPLE: A schema with a target namespace is including a schema without a target namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:ns="http://www.example.org"
        targetNamespace="http://www.example.org">
    <!-- the including schema -->
    <xsd:include schemaLocation="included.xsd"/>
:
</xsd:schema>

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <!—this is the included schema -->
:
</xsd:schema>
```

Will result the TTCN-3 modules (please note, the content of the modules may come from more than one schemas).

```
module http_www_example_org {
: // contains definitions mapped from both schemas
}
with {
    encode "XML";
    variant "namespace as 'http://www.example.org' prefix 'ns1'";
}

module NoTargetNamespace {
: // contains definitions mapped from the schema without target namespace only
}
with {
    encode "XML"
}
```

5.1.3 Imports

All XSD import statements (i.e. *import* element information items and the related *xmlns* attributes, where present) shall be mapped to equivalent TTCN-3 *import* statements, importing all definitions from the other TTCN-3 module. All XSD components are **`public`** by default (see clause 8.2.3 of ETSI ES 201 873-1 [1]). Multiple XSD *import* element information items with the same *namespace* attribute (including no target namespace) shall be mapped to a single TTCN-3 import statement.

NOTE 1: The above statement means that XSD components using imported XSD references are complete, i.e. in case of implicit mapping it is not needed to additionally import the schema containing the referenced XSD components to TTCN-3, unless the referenced XSD component wanted to be used in TTCN-3 directly.

NOTE 2: XSD permits a bare <import> information item (in schemas having a target namespace). This allows unqualified references to foreign components with no target namespace without giving hints where to find them. The resolution of such cases is left to tool implementations. It is allowed to import single XSD components into TTCN-3. When the TTCN-3 import statement is importing single definitions or definitions of the same kind from XSD (see clauses 8.2.3.2 and 8.2.3.4 of ETSI ES 201 873-1 [1]), or an import all statement contains an exception list (see clause 8.2.3.5 of ETSI ES 201 873-1 [1]), this results in the import of a **`type`** definition only, but not in the import of a **`group`**, **`template`**, **`testcase`**, etc.

NOTE 3: Please note that importing all types of a target namespace has the same effect as importing all definitions of that namespace (i.e. "**import from** *TargetNamespace* **{ type all }**;" results in the same as "**import from** *TargetNamespace* **all;**").

It is not allowed to import XSD import statements to TTCN-3 (i.e. there is no transitive import of XSD import statements as defined for TTCN-3, see clause 8.2.3.7 of ETSI ES 201 873-1 [1]).

5.1.4 Attributes of the XSD schema element

If the TTCN-3 module corresponds to a (present) target namespace and the value of the *attributeFormDefault* and/or *elementFormDefault* attributes of any *schema* element information items that contribute to the given TTCN-3 module is *qualified*, the encoding instructions **"attributeFormQualified"** and/or **"elementFormQualified"** shall be attached accordingly to the given TTCN-3 module. All fields of TTCN-3 definitions in the given TTCN-3 module corresponding to local *attribute* declarations or to attribute and *attributeGroup* references in *schema* element information items with the value of its *attributeFormDefault* attribute being *unqualified* (explicitly or implicitly via defaulting) shall be supplied with the **"form as unqualified"** encoding instruction, unless a *form* attribute of the given declaration requires differently (see clause 7.1.6). All fields of TTCN-3 definitions in the given TTCN-3 module corresponding to local *element* declarations or element and model *group* references in *schema* element information items with the value of its *elementFormDefault* attribute *unqualified* (explicitly or implicitly via defaulting) shall be supplied with the **"form as unqualified"** encoding instruction, unless a *form* attribute of the given declaration requires differently (see clause 7.1.6).

Mapping of the *blockDefault* XSD attribute information item see in clauses 7.1.10, 8.1 and 8.2.

The *finalDefault*, *id*, *version* and *xml:lang* attributes of schema elements shall be ignored.

EXAMPLE: Mapping of schema attributes:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.example.org"
            attributeFormDefault="qualified"
            elementFormDefault="unqualified">
    <xsd:complexType name="CType1">
        <xsd:sequence>
            <xsd:element name="elem" type="xsd:integer"/>
        </xsd:sequence>
        <xsd:attribute name="attrib" type="xsd:integer"/>
    </xsd:complexType></xsd:schema>

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.example.org"
            attributeFormDefault="unqualified"
            elementFormDefault="qualified">
    <xsd:complexType name="CType2">
        <xsd:sequence>
            <xsd:element name="elem" type="xsd:integer"/>
        </xsd:sequence>
        <xsd:attribute name="attrib" type="xsd:integer"/>
    </xsd:complexType>
</xsd:schema>
```

Will result in the TTCN-3 modules (please note, that the content of the modules may come from more than one schemas).

```
module http_www_example_org {
    type record CType1 {
        XSD.Integer attrib optional,
        XSD.Integer elem
    }
    with {
        variant(attrib)"attribute";
        variant(elem)"form as unqualified";
    }

    type record CType2 {
        XSD.Integer attrib optional,
        XSD.Integer elem
    }
    with {
```

```
        variant(attrib)"attribute";
        variant(attrib)"form as unqualified";
    }
}
with {
    encode "XML";
    variant "namespace as 'http://www.example.org'";
    variant "attributeFormQualified";
    variant "elementFormQualified";
}
```

5.1.5 The control namespace

The control namespace is the namespace of the schema-instance attributes defined in clause 2.6 of XSD Part 1 [9], for direct use in any XML documents (e.g. in the special XML attribute value "xsi:nil", see mapping of the *nillable* XSD attribute in clause 7.1.11 or in case of substitutable types is the special XML attribute value "xsi:type", see clause 8.2, etc.). It shall be specified globally, with the controlNamespace encoding instruction attached to the TTCN-3 module.

NOTE 1: These attributes are in the namespace http://www.w3.org/2001/XMLSchema-instance.

NOTE 2: See also the definition "**xsi: attributes**" in clause 3.1 of the present document.

EXAMPLE: Identifying the control namespace of a module:

```
module MyModule
{
:
}
with {
    encode "XML";
    variant "controlNamespace'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

5.2 Name conversion

5.2.1 General

Translation of identifiers (e.g. type or field names) has a critical impact on the usability of conversion results: primarily, it shall guarantee TTCN-3 consistency, but, in order to support migration of conversion results from code generated with tools based on Recommendation ITU-T X.694 [4], it shall also generate identifiers compatible with that standard. It shall also support portability of conversion results (the TTCN-3 code and the encoding instruction set) between TTCN-3 tools of different manufacturers, which is only possible if identifier conversion is standardized.

For different reasons a valid XSD identifier may not be a valid identifier in TTCN-3. For example, it is valid to specify both an attribute and an element of the same name in XSD. When mapped in a naïve fashion, this would result in two different types with the same name in TTCN-3.

A name conversion algorithm has to guarantee that the translated identifier name:

a) is unique within the scope it is to be used;

b) contains only valid characters;

c) is not a TTCN-3 keyword or a keyword used in an extension package;

d) is not a reserved word (e.g. "base" or "content").

The present document specifies the generation of:

a) TTCN-3 type reference names corresponding to the names of model group definitions, top-level element declarations, top-level attribute declarations, top-level complex type definitions, and user-defined top-level simple type definitions;

b) TTCN-3 identifiers corresponding to the names of top-level element declarations, top-level attribute declarations, local element declarations, and local attribute declarations;

c) TTCN-3 identifiers for the mapping of certain simple type definitions with an enumeration facet (see clause 6.1.5);

d) TTCN-3 identifiers of certain sequence components introduced by the mapping (see clause 7);

e) TTCN-3 module names corresponding to the target namespaces of the XSD documents being translated.

All of these TTCN-3 names shall be generated by applying clause 5.2.2 either to the name of the corresponding schema component, or to a member of the value of an enumeration facet, or to a specified character string, as specified in the relevant clauses of the present document.

5.2.2 Name conversion rules

Names of attribute declarations, element declarations, model group definitions, user-defined top-level simple type definitions, and top-level complex type definitions can be identical to TTCN-3 reserved words, can contain characters not allowed in TTCN-3 identifiers. In addition, there are cases in which TTCN-3 names are required to be distinct where the names of the corresponding XSD schema components (from which the TTCN-3 names are mapped) are allowed to be identical.

First:

a) the character strings to be used as names in a TTCN-3 module, shall be ordered in accordance to clause 5.2.3 (i.e. primary ordering the character strings according to their categories as names of elements, followed by names of attributes, followed by names of type definitions, followed by names of model groups, and subsequently ordering in alphabetical order);

NOTE 1: "The character strings to be used as names in a TTCN-3 module" refers to the character strings received after applying specific transformations described in relevant clauses of the present document (which still might need to be converted before written to the module by tools).

NOTE 2: The above ordering of character strings is necessary to produce the same final names for the same definitions independent of the order in which tools are processing *schema* elements with the same target namespace. It does not affect the order in which the generated TTCN-3 definitions are written to the modules by tools.

Target namespace values used in XSD schema documents shall be ordered alphabetically, independently from the above components (after conversion they are merely used as TTCN-3 module names). The string value of the target namespace values shall be used, i.e. without un-escaping or removing trailing "/" SOLIDUS characters of the authority part or any other changes to the character string.

NOTE 3: "Namespaces in XML 1.0 (Second Edition)" [6], clause 2.3 defines that namespaces are treated as strings and are identical only if they are the same character strings; for example, the target namespaces http://www.example.org and http://www.example.org/ or http://www.example.org/~wilbur and http://www.example.org/%7ewilbur are different and all result in different TTCN-3 module names.

Secondly, the following character substitutions shall be applied, in order, to each character string being mapped to a TTCN-3 name, where each substitution (except the first) shall be applied to the result of the previous transformation:

b) the characters " " (SPACE), "**.**" (FULL STOP), "-" (HYPEN-MINUS), ":" (COLON) and "/" (SOLIDUS) shall all be replaced by a "_" (LOW LINE);

NOTE 4: Please note that the " " (SPACE), ":" (COLON) and "/" (SOLIDUS) character may appear in (target) namespace attributes only but not in local parts of XML qualified names; i.e. the colon above does not refer to the colon separating the Prefix and the NCName parts of XML qualified names (see [9], clause 3.2.18).

c) any character except "A" to "Z" (LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z), "a" to "z" (LATIN SMALL LETTER A to LATIN SMALL LETTER Z), "0" to "9" (DIGIT ZERO to DIGIT NINE), and "_" (LOW LINE) shall be removed;

d) a sequence of two or more "_" (LOW LINE) characters shall be replaced with a single "_" (LOW LINE);

e) "_" (LOW LINE) characters occurring at the beginning or at the end of the name shall be removed, except trailing "_" (LOW LINE) characters resulted from converting target namespace values (to be used as TTCN-3 module names);

f) if a character string that is to be used as a name of a TTCN-3 type starts with a lower-case letter, the first letter shall be capitalized (converted to upper-case); if it starts with a digit (DIGIT ZERO to DIGIT NINE), it shall be prefixed with an "**X**" (LATIN CAPITAL LETTER X) character;

g) if a character string that is to be used as an identifier of a structured type field or enumeration value starts with an upper-case letter, the first letter shall be uncapitalized (converted to lower-case); if it starts with a digit (DIGIT ZERO to DIGIT NINE), it shall be prefixed with an "**x**" (LATIN SMALL LETTER X) character;

h) if a character string that is to be used as a name of a TTCN-3 type definition or as a type reference name is empty, it shall be replaced by "**X**" (LATIN CAPITAL LETTER X); and

i) if a character string that is to be used a name of a record or union field or enumeration value is empty, it shall be replaced by "**x**" (LATIN SMALL LETTER X).

EXAMPLE 1: Simple character substitutions:

- **rule b:** `"TTCN-3"` → `"TTCN_3"`
- **rules b, c and d:** `"TTCN-+-3"` → `"TTCN_3"`
- **rules b and f (for types):** `"ac/dc"` → `"Ac_dc"`
- **rule f (for types):** `"007"` → `"X007"`
- **rules b and g (e.g. enum value):** `"0-value"` → `"x0_value"`

Finally, depending on the kind of name being generated, one of the following four items shall apply:

j) If the name being generated is the name of a TTCN-3 module and the character string generated by items a) to e) above is identical to an another, previously generated TTCN-3 module name, then a postfix shall be appended to the character string: the postfix shall consist of a "_" (LOW LINE) followed by the canonical lexical representation (see W3C® XML Schema Part 2 [9], clause 2.3.1) of an integer, unless the name already finishes with a "_" (LOW LINE) character, in which case the postfix is an integer only. This integer shall be the least positive integer such that the new name is different from all previously generated TTCN-3 modules names and clashing definition name.

k) If the name being generated is the name of a TTCN-3 type and the character string generated by items a) to i) above is identical to the name of another TTCN-3 type previously generated in the same TTCN-3 module, or is identitcal to the name of the TTCN-3 module generated for its target namespace or one of the modules imported into that module or is one of the reserved words specified in clause 11.27 of Recommendation ITU-T X.680 [3], then a postfix shall be appended to the character string generated according to the above rules. The postfix shall consist of a "_" (LOW LINE) followed by the canonical lexical representation (see W3C® XML Schema Part 2 [9], clause 2.3.1) of an integer. This integer shall be the least positive integer such that the new name is different from the type reference name of any other TTCN-3 type assignment previously generated in any of those TTCN-3 modules.

l) If the name being generated is the identifier of a field of a record or a union type, and the character string generated by the rules in items a) to i) above is identical to the identifier of a previously generated field identifier of the same type or in case of a union type if it is identical to any identifier from the list of effective fields (see clause 6.2.5.0 of ETSI ES 201 873-1 [1]), then a postfix shall be appended to the character string generated by the above rules. The postfix shall consist of a "_" (LOW LINE) followed by the canonical lexical representation (see W3C® XML Schema Part 2 [9], clause 2.3.1) of an integer. This integer shall be the least positive integer such that the new identifier is different from the identifier of any previously generated component of that sequence, set, or choice type. Field names that are one of the TTCN-3 keywords (see clause A.1.5 of ETSI ES 201 873-1 [1]), keywords used in extension packages or names of predefined functions (see clause 16.1.2 of ETSI ES 201 873-1 [1]) after applying the postfix to clashing field names, shall be suffixed by a single "_" (LOW LINE) character.

NOTE 5: ETSI ES 201 873-1 [1] clause A.1.5 table A.2 defines the keywords of the core language and table A.5 defines the keywords used in extension packages. However, other language mappings and TTCN-3 language extensions (see [i.5] to [i.12] and [i.13]) may also be published after the publication of the present document) may define additional keywords and rules for handling those keywords in TTCN-3 modules requiring the given extension.

m) If the name being generated is the identifier of an enumeration item (see clause 6.2.4 of ETSI ES 201 873-1 [1]) of an enumerated type, and the character string generated by the rules in items a) to i) above is identical to the identifier of another enumeration item previously generated in the same enumerated type, then a postfix shall be appended to the character string generated by the above rules. The postfix shall consist of a "_" (LOW LINE) followed by the canonical lexical representation (see W3C® XML Schema Part 2 [9], clause 2.3.1) of an integer. This integer shall be the least positive integer such that the new identifier is different from the identifier in any other enumeration item already present in that TTCN-3 enumerated type. Enumeration names that are one of the TTCN-3 keywords (see clause A.1.5 of ETSI ES 201 873-1 [1]), keywords used in extension packages or names of predefined functions (see clause 16.1.2 of ETSI ES 201 873-1 [1]) after applying the postfix to clashing enumeration names, shall be suffixed by a single "_" (LOW LINE) character.

EXAMPLE 2: Conversion of an XML Schema composed of two schema elements with identical target namespaces:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.example.org/1">
    <!— this file is: includeCircular1a.xsd -->
    <xsd:include schemaLocation="includeCircular1b.xsd"/>
    <!-- simpleType "Foobar" -->
    <xsd:simpleType name="Foobar">
        <xsd:restriction base="xsd:integer"/>
    </xsd:simpleType>
    <!-- attribute "Foo-Bar" -->
    <xsd:attribute name="Foo-Bar" type="xsd:integer"/>
    <!-- attribute "Foo_Bar" -->
    <xsd:attribute name="Foo_Bar" type="xsd:integer"/>
    <!-- attribute "Foobar" -->
    <xsd:attribute name="Foobar" type="xsd:integer"/>
    <!-- element "foobar" -->
    <xsd:element name="foobar" type="xsd:integer"/>
    <!-- element "Foobar" -->
    <xsd:element name="Foobar" type="xsd:integer"/>
    <xsd:complexType name="Akarmi">
        <xsd:sequence/>
            <!-- complexType attribute "foobar" -->
            <xsd:attribute name="foobar" type="xsd:integer"/>
            <!-- complexType attribute "Foobar" -->
            <xsd:attribute name="Foobar" type="xsd:integer"/>
    </xsd:complexType>
</xsd:schema>

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.example.org/1">
    <!-- this file is: includeCircular1b.xsd -->
    <xsd:include schemaLocation="includeCircular1a.xsd"/>
    <!-- simpleType "foobar" -->
    <xsd:simpleType name="foobar">
        <xsd:restriction base="xsd:integer"/>
    </xsd:simpleType>
    <!-- attribute "foobar" -->
    <xsd:attribute name="foobar" type="xsd:integer"/>
</xsd:schema>
```

Will be translated to:

```
module http_www_example_org_1 {
/* this file is: includeCircular1a.xsd */
    /* simpleType "Foobar" */
    type XSD.Integer Foobar_4
    // postfixed with "_4" as types are the third category and capital letters are preceding
    // small letters in ISO 646.
    with {
        variant "name as 'Foobar'";
    }
```

```
    /* attribute "Foo-Bar" */
    type XSD.Integer Foo_Bar
    with {
        variant "name as 'Foo-Bar'"; variant "attribute";
    }

    /* attribute "Foo_Bar" */
    type XSD.Integer Foo_Bar_1
    // postfixed with "_1" as after changing dash to underscore in the name of the attribute
    // "Foo-Bar", the names of the two types are clashing with each other.
    with {
        variant "name as 'Foo_Bar'"; variant "attribute";
    }

    /* attribute "Foobar" */
    type XSD.Integer Foobar_2
    // postfixed with "_2" as attributes are the second category and capital letters are
    // preceding small letters in ISO 646.
    with {
        variant "name as 'Foobar'";
        variant "attribute";
    }

    /* element "foobar" */
    type XSD.Integer Foobar_1
    // postfixed with "_1" as elements are the first category and small letters are following
    // capital letters in ISO 646.
    with {
        variant "name as 'foobar'";
        variant "element";
    }

    /* element "Foobar" */
    type XSD.Integer Foobar
    // no postfix as elements are the first category and capital letters are preceding
    // small letters in ISO 646.
    with {
        variant "element";
    }

    type record Akarmi {
        /* complexType attribute "Foobar" */
        XSD.Integer foobar optional,
        /* complexType attribute "foobar" */
        XSD.Integer foobar_1 optional;
    }
    with {
        variant (foobar) "name as capitalized";
        variant (foobar_1) "name as 'foobar'";
        variant (foobar,foobar_1) "attribute";
}

    /* this file is: includeCircular1b.xsd*/
    /* simpleType "foobar" */
    type XSD.Integer Foobar_5
    // postfixed with "_5" as types are the third category and small letters are following
    // capital letters in ISO 646.
    with {
        variant "name as 'foobar'";
    }

    /* attribute "foobar" */
    type XSD.Integer Foobar_3
    // postfixed with "_3" as attributes are the second category and small letters are
    // following capital letters in ISO 646.
    with {
        variant "name as 'foobar'";
        variant "attribute";
    }
}
with {
    variant "namespace as 'http_www.example.org/1'";
}
```

For an TTCN-3 type definition name or field identifier that is generated by applying this clause to the name of an element declaration, attribute declaration, top-level complex type definition or user-defined top-level simple type definition, if the type definition name generated is different from the value of the *name* attribute of the corresponding schema component, a final "name as…" variant attribute shall be attached to the TTCN-3 type definition with that type definition name (or to the field with that identifier) as specified in the items below:

a) If the only difference is the case of the first letter (which is upper case in the type definition name and lower case in the *name*), then the variant attribute `"name as uncapitalized"` shall be used.

b) If the only difference is the case of the first letter (which is lower case in the identifier and upper case in the *name*), then the variant attribute `"name as capitalized"` shall be applied to the field concerned or the "name all as capitalized" shall be applied to the related type definition (in this case the attribute has effect on all identifiers of all fields but not on the name of the type!).

c) Otherwise, the `"name as '<name>'"` variant attribute shall be used, where *<name>* is the value of the corresponding *name* attribute.

EXAMPLE 3: Using the "**name**" variant attribute:

The top-level complex type definition:

```
<xsd:complexType name="COMPONENTS">
    <xsd:sequence>
        <xsd:element name="Elem" type="xsd:boolean"/>
        <xsd:element name="elem" type="xsd:integer"/>
        <xsd:element name="Elem-1" type="xsd:boolean"/>
        <xsd:element name="elem-1" type="xsd:integer"/>
    </xsd:sequence>
</xsd:complexType>
```

Will be mapped to the TTCN-3 type assignment e.g. as:

```
type record COMPONENTS_1
{
    boolean elem,
    integer elem_1,
    boolean elem_1_1,
    integer elem_1_2
}
with {
    variant "name as 'COMPONENTS'";
    variant (elem) "name as capitalized";
    variant (elem_1) "name as 'elem'";
    variant (elem_1_1) "name as 'Elem-1'";
    variant (elem_1_2) "name as 'elem-1'";
};
```

For an TTCN-3 identifier that is generated by applying this clause for the mapping of a simple type definition with an enumeration facet where the identifier of the generated TTCN-3 enumeration value is different from the corresponding member of the value of the *enumeration* facet, a "text as…" variant attribute shall be assigned to the TTCN-3 enumerated type, with qualifying information specifying the identifier of the enumeration item of the enumerated type. One of the two following items shall apply:

a) If the only difference is the case of the first letter (which is lower case in the identifier and upper case in the member of the value of the *enumeration* facet), then the "text "*<TTCN-3 enumeration identifier>*" as capitalized" variant attribute shall be used.

b) If all TTCN-3 enumeration values differ in the case of the first letter only (which is lower case in the identifier and upper case in the member of the value of the *enumeration* facet), then the "text all as capitalized" variant attribute shall be used.

c) Otherwise, the "text "*<TTCN-3 enumeration identifier>*" as "*<member of the* value *of the enumeration facet>*"" variant attribute shall be used.

EXAMPLE 4: Using the "text as…" variant attribute:

The XSD enumeration facet:

```
<xsd:simpleType name="state">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Off"/>
        <xsd:enumeration value="off"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to the TTCN-3 type assignment:

```
type enumerated State { off, off_1 }
with {
    variant "name as uncapitalized";
    variant "text 'off' as capitalized";
    variant "text 'off_1' as 'off'";
}
```

5.2.3 Order of the mapping

An order shall be imposed on the top-level schema components of the source XSD Schema on which the mapping is performed. This applies to model group definitions, top-level complex type definitions, user-defined top-level simple type definitions, top-level attribute declarations, and top-level element declarations.

NOTE: Other top-level schema components are not mapped to TTCN-3, and XSD built-in data types are mapped in a special way.

The order is specified in the three following items:

a) Top-level schema components shall first be ordered by their original target namespace (i.e. use the order produced by item a) of clause 5.2.2), with the absent namespace preceding all namespace names in ascending alphabetical order.

b) Within each target namespace, top-level schema components shall be divided into four sets ordered as follows:

   1) element declarations;

   2) attribute declarations;

   3) complex type definitions and simple type definitions;

   4) model group definitions.

c) Within each set of item b), schema components shall be ordered by name in ascending alphabetical order.

TTCN-3 type definitions that correspond directly to the XSD schema components shall be generated in the order of the corresponding XSD schema components.

5.3 Mapping of XSD schema components

Table 1a: Mapping of XSD schema components

XSD schema component	Sub-category	W3C® XML Schema reference	TTCN-3 mapping defined by
attribute declaration		Part 1, 3.2 [8]	Clause 7.4
element declaration	global	Part 1, 3.3 [8]	Clause 7.3
	local		Clause 7.3
	head of a substitution group		Clause 8.1.1
complex type definition	not substitutable	Part 1, 3.4 [8]	Clause 7.6
	substitutable		Clause 8.2
built-in datatypes		Part 2 [9]	Clause 6
attribute use		Part 1, 3.5 [8]	Clause 7.1.12
attribute group definition		Part 1, 3.6 [8]	Clause 7.4.2
model group definition		Part 1, 3.7 [8]	Clause 7.9
model group use		Part 1, 3.8 [8]	Clause 7.6.7

XSD schema component	Sub-category	W3C® XML Schema reference	TTCN-3 mapping defined by
particle		Part 1, 3.9 [8]	Clauses 7.6.4, 7.6.5 and 7.6.6
wildcard		Part 1, 3.10 [8]	Clause 7.1.15
identity-constraint definition		Part 1, 3.11 [8]	Clause 7.10
notation declaration		Part 1, 3.12 [8]	*ignored by the mapping*
annotation		Part 1, 3.13 [8]	*ignored by the mapping*
simple type definition	not substitutable	Part 1, 3.14 [8]	Clause 7.5
	substitutable		Clause 8.2
schema		Part 1, 3.15 [8]	Clause 7.2
ordered		Part 2, 4.2.2.1 [9]	*ignored by the mapping*
bounded		Part 2, 4.2.3.1 [9]	*ignored by the mapping*
cardinality		Part 2, 4.2.4.1 [9]	*ignored by the mapping*
numeric		Part 2, 4.2.5.1 [9]	*ignored by the mapping*
length		Part 2, 4.3.1.1 [9]	Clause 6.1.1
minLength		Part 2, 4.3.2.1 [9]	Clause 6.1.2
maxLength		Part 2, 4.3.3.1 [9]	Clause 6.1.3
pattern		Part 2, 4.3.4.1 [9]	Clause 6.1.4
enumeration		Part 2, 4.3.5.1 [9]	Clause 6.1.5
whiteSpace		Part 2, 4.3.6.1 [9]	Clause 6.1.6
maxInclusive		Part 2, 4.3.7.1 [9]	Clause 6.1.8
maxExclusive		Part 2, 4.3.8.1 [9]	Clause 6.1.10
minExclusive		Part 2, 4.3.9.1 [9]	Clause 6.1.9
minInclusive		Part 2, 4.3.10.1 [9]	Clause 6.1.7
totalDigits		Part 2, 4.3.11.1 [9]	Clause 6.1.11
fractionDigits		Part 2, 4.3.12.1 [9]	*ignored by the mapping*

5.4 Unsupported features

XSD and TTCN-3 are very distinct languages. Therefore some features of XSD have no equivalent in TTCN-3 or make no sense when translated to the TTCN-3 language. Whenever possible, these features are translated into encoding instructions completing the TTCN-3 code. The following list contains a collection of the unsupported features:

a) Numeric types are not allowed to be restricted by patterns.

b) List types are not allowed to be restricted by enumerations or patterns.

c) Specifying the number of fractional digits for float types is not supported.

d) Translation of the identity-constraint definition schema components (*unique*, *key*, *keyref*, *selector* and *field* elements) are not supported.

e) All time types (see clause 6.5) restrict year to 4 digits.

5.5 Conformance and compatibility

For an implementation claiming to support the use of XSD with TTCN-3, 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].

The language mapping presented in the present document is compatible to:

- ETSI ES 201 873-1 [1], V4.2.1.

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

6 Built-in data types

6.0 General

XSD built-in data types may be primitive or derived types. The latter are gained from primitive types by means of a restriction mechanism called facets. For the mapping of primitive types, a specific TTCN-3 module *XSD* is provided by the present document, which defines the relation of XSD primitive types to TTCN-3 types. Whenever a new *simpleType* is defined, with a built-in XSD type as its base type, it shall be mapped directly from types defined in the module XSD.

EXAMPLE:

```
<xsd:simpleType name="e1">
    <xsd:restriction base="xsd:integer"/>
</xsd:simpleType>
```

Becomes:

```
type XSD.Integer E1
with {
    variant "name as uncapitalized";
}
```

In the following clauses both the principle mappings of facets and the translation of primitive types are given. The complete content of the XSD module is given in annex A.

6.1 Mapping of facets

6.1.0 General

Table 2 summarizes the facets for the built-in types that are mapped to TTCN-3 specifically, i.e. to a specific TTCN-3 language construct. Facets, allowed by XML Schema but without a counterpart in TTCN-3, shall be retained by a "transparent" encoding instruction as given in clause 6.1.13 and therefore not marked in table 2.

Table 2: Mapping support for facets of built-in types

Facet / Type	length	min Length	max Length	pattern	enum.	min Incl.	max Incl.	min Excl.	max Excl.	total Digits	white Space
string	✓ (see note 1)	✓ (see note 2)	✓ (see note 2)	✓ (see note 2)	✓						✓ (see note 3)
integer					✓	✓	✓	✓	✓	✓	
float					✓	✓	✓	✓	✓	✓ (see note 4)	
time				✓	✓						
list	✓	✓	✓								
boolean											

NOTE 1: With the exception of *QName* which does not support length restriction.
NOTE 2: With the exception of *hexBinary* which does not support patterns.
NOTE 3: With the exception of some types (see clause 6.1.6).
NOTE 4: With the exception of *decimal* which does support *totalDigits*.

6.1.1 Length

The XSD facet *length* describes, how many units of length a value of the given simple type shall have. For *string* and data types derived from *string*, *length* is measured in units of characters. For *hexBinary* and *base64Binary* and data types derived from them, *length* is measured in octets. For data types derived by *list*, *length* is measured in number of list items. A length-restricted XSD type shall be mapped to a corresponding length restricted TTCN-3 type.

EXAMPLE 1:

```
<xsd:simpleType name="e2">
    <xsd:restriction base="xsd:string">
        <xsd:length value="10"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is translated to the following TTCN-3 type:

```
type XSD.String E2 length(10)
with {
    variant "name as uncapitalized";
}
```

For built-in list types (see clause 6.6) the number of elements of the resulting structure will be restricted.

EXAMPLE 2:

```
<xsd:simpleType name="e3">
    <xsd:restriction base="xsd:NMTOKENS">
        <xsd:length value="10"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to TTCN-3 e.g. as:

```
type XSD.NMTOKENS E3 length(10)
with {
    variant "name  as uncapitalized";
}
```

6.1.2 MinLength

The XSD facet *minLength* describes the minimal length that a value of the given simple type shall have. It shall be mapped to a *length* restriction in TTCN-3 with a set lower bound and an open upper bound. The *fixed* XSD attribute (see clause 7.1.5) shall be ignored.

EXAMPLE:

```
<xsd:simpleType name="e4">
    <xsd:restriction base="xsd:string">
        <xsd:minLength value="3"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is translated to TTCN-3 e.g. as:

```
type XSD.String E4 length(3 .. infinity)
with {
    variant "name as uncapitalized";
}
```

6.1.3 MaxLength

The XSD facet *maxLength* describes the maximal length that a value of the given simple type shall have. It shall be mapped to a *length* restriction in TTCN-3 with a set upper bound and a lower bound zero. The *fixed* XSD attribute (see clause 7.1.5) shall be ignored.

EXAMPLE:

```
<xsd:simpleType name="e5">
    <xsd:restriction base="xsd:string">
        <xsd:maxLength value="5"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to TTCN-3 e.g. as:

```
type XSD.String E5 length(0 .. 5)
with {
    variant "name  as uncapitalized";
}
```

6.1.4 Pattern

The XSD *pattern* facet allows constraining the value space of XSD data types by restricting the value notation by a regular expression. This facet is supported for XSD types derived directly or indirectly from the XSD string type. For these types pattern facets shall directly be mapped to TTCN-3 pattern subtyping. As the syntax of XSD regular patterns differs from the syntax of the TTCN-3 pattern subtyping, a mapping of the pattern expression has to be applied. The symbols "(" (LEFT PARENTHESIS), ")" (RIGHT PARENTHESIS), "|" (VERTICAL LINE), "[" (LEFT SQUARE BRACKET), "]" (RIGHT SQUARE BRACKET) and "^" (CIRCUMFLEX ACCENT) shall not be changed and shall be translated directly. Other meta characters shall be mapped according to tables 3 and 4.

Table 3: Translation of meta characters

XSD	TTCN-3
.	?
\s	[\q{0,0,0,20}\q{0,0,0,10}\t\r] (see note)
\S	[^\q{0,0,0,20}\q{0,0,0,10}\t\r] (see note)
\d	\d
\D	[^\d]
\w	\w
\W	[^\w]
\i	[\w\d:]
\I	[^\w\d:]
\c	[\w\d.\-_:]
\C	[^\w\d.\-_:]
NOTE: \q{0,0,0,20} denotes the " " (SPACE) graphical character and \q{0,0,0,10} denotes the line feed (LF) control character.	

Table 4: Translation of quantifiers

XSD	TTCN-3
?	#(0,1)
+	#(1,)
*	#(0,)
{n,m}	#(n,m)
{n}	#n
{n,}	#(n,)

Unicode characters in XSD patterns are directly translated but the syntax changes from `&#x`**`gprc;`** in XSD to `\q`**`{g, p, r, c}`** in TTCN-3, where **g**, **p**, **r**, and **c** each represent a single character.

Escaped characters in XSD shall be mapped to the appropriate character in TTCN-3 (e.g. ".", and "+") or, if this character has a meta-character meaning in TTCN-3 patterns, to an escaped character in TTCN-3. The double quote character shall be mapped to a pair of double quote characters in TTCN-3. Character categories and blocks (like `\p`**`{`**`Lu`**`}`** or `\p`**`{`**`IsBasicLatin`**`}`**) are not supported. The mapping shall result in a valid TTCN-3 pattern according to clause B.1.5 of ETSI ES 201 873-1 [1].

EXAMPLE:

```
<xsd:simpleType name="e6">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="(aUser|anotherUser)@(i|I)nstitute"/>
    </xsd:restriction>
</xsd:simpleType>
```

Will be mapped to the following TTCN-3 expresion:

```
type XSD.String E6 (pattern "(aUser|anotherUser)@(i|I)nstitute")
with {
    variant "name as uncapitalized";
}
```

6.1.5 Enumeration

The facet *enumeration* constraints the value space of XSD simple types to a specified set of values.

A simple type definition that is derived from an XSD string type (directly or indirectly) by *restriction* using the *enumeration* facet, shall be mapped to a TTCN-3 **`enumerated`** type (see clause 6.2.4 of ETSI ES 201 873-1 [1]), where each XSD *enumeration* information item is mapped to a TTCN-3 enumeration value of a TTCN-3 enumerated type (see clause 6.2.4 of ETSI ES 201 873-1 [1]), as follows:

a) For each member of the XSD enumeration facet, a TTCN-3 enumeration item shall be added to the enumerated type that is an identifier (i.e. without associated integer value), except for members not satisfying a relevant length, minLength, maxLength, pattern facet or a whiteSpace facet with a value of replace or collapse and the member name contain any of the characters HORIZONTAL TABULATION, NEWLINE or CARRIAGE RETURN, or (in the case of collapse) contain leading, trailing, or multiple consecutive SPACE characters.

b) Each enumeration identifier shall be generated by applying the rules defined in clause 5.2.2 of the present document to the corresponding value of the enumeration facet.

c) The members of the same enumeration facet (children of the sameXSD *restriction* element) shall be mapped in ascending lexicographical order and any duplicate members shall be discarded.

NOTE 1: For further information on XSD datatypes derived from the XSD string type see the the built-in datatype hierarchy in clause 3 of XML Schema Part 2 [9].

A simple type definition that is derived from the XSD integer type (directly or indirectly) by restriction using the *enumeration* facet, shall be mapped to a TTCN-3 **`enumerated`** type (see clause 6.2.4 of ETSI ES 201 873-1 [1]), where each XSD *enumeration* information item is mapped a TTCN-3 enumeration value, as specified below. In this case the enumeration names are artificial and the encoded XML component shall contain the integer values, not the TTCN-3 enumeration names. The encoder shall be instructed to do so with the encoding instruction "useNumber":

a) For each member of the XSD enumeration facet, a TTCN-3 enumeration item shall be added to the enumerated type that is an enumeration identifier plus the associated integer value shall be added to the enumeration type, except for facet values not satisfying a relevant length, minLength, maxLength, pattern facet or a whiteSpace facet with a value of replace or collapse and the member name contain any of the characters HORIZONTAL TABULATION, NEWLINE or CARRIAGE RETURN, or (in the case of collapse) contain leading, trailing, or multiple consecutive SPACE characters.

b) The identifier of each enumeration item shall be generated by concatenating the character string "int" with the canonical lexical representation (see W3C® XML Schema Part 2 [9], clause 2.3.1) of the corresponding member of the value of the enumeration facet. The assigned integer value shall be the TTCN-3 integer value notation for the member.

c) The members of the same enumeration facet (children of the sameXSD *restriction* element) shall be mapped in ascending numerical order and any duplicate members shall be discarded.

NOTE 2: For further information on XSD datatypes derived from the XSD integer type see the the built-in datatype hierarchy in clause 3 of XML Schema Part 2 [9].

Any other enumeration facet shall be mapped to value list subtyping, if this is allowed by ETSI ES 201 873-1 [1], that is either a single value or a union of single values corresponding to the members of the enumeration facet. If a corresponding value list subtyping is not allowed by ETSI ES 201 873-1 [1], the enumeration facet shall be ignored.

NOTE 3: The **enumeration** facet applies to the value space of the **base type definition**. Therefore, for an **enumeration** of the XSD built-in datatypes **QName**, the value of the **uri** component of the use_qname **record** (see clause 6.6.4) is determined, in the XML representation of an XSD Schema, by the namespace declarations whose scope includes the **QName**, and by the prefix (if any) of the **QName**.

EXAMPLE 1: The following represents a user-defined top-level simple type definition that is a restriction of xsd:string with an enumeration facet:

```
<xsd:simpleType name="state">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="off"/>
        <xsd:enumeration value="on"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to the TTCN-3 type definition:

```
type enumerated State {off, on_ }
with {
    variant "name as uncapitalized";
    variant "text 'on_' as 'on'";
}
```

EXAMPLE 2: The following represents a user-defined top-level simple type definition that is a restriction of xsd:integer with an enumeration facet:

```
<xsd:simpleType name="integer-0-5-10">
    <xsd:restriction base="xsd:integer">
        <xsd:enumeration value="0"/>
        <xsd:enumeration value="5"/>
        <xsd:enumeration value="-5"/>
        <xsd:enumeration value="10"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to the TTCN-3 type definition:

```
type enumerated Integer_0_5_10 {int_5(-5), int0(0), int5(5), int10(10)}
with {
    variant "name as 'integer-0-5-10'";
    variant "useNumber";
}
```

EXAMPLE 3: The following represents a user-defined top-level simple type definition that is a restriction of xsd:integer with a minInclusive and a maxInclusive facet:

```
<xsd:simpleType name="integer-1-10">
    <xsd:restriction base="xsd:integer">
        <xsd:minInclusive value="1"/>
        <xsd:maxInclusive value="10"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to the TTCN-3 type definition:

```
type integer Integer_1_10 (1..10)
with {
    variant "name as integer-1-10";
}
```

EXAMPLE 4: The following represents a user-defined top-level simple type definition that is a restriction (with a minExclusive facet) of another simple type definition, derived by restriction from xsd:integer with the addition of a minInclusive and a maxInclusive facet:

```
<xsd:simpleType name="multiple-of-4">
    <xsd:restriction>
        <xsd:simpleType>
            <xsd:restriction base="xsd:integer">
                <xsd:minInclusive value="1"/>
                <xsd:maxInclusive value="10"/>
            </xsd:restriction>
        </xsd:simpleType>
        <xsd:maxExclusive value="5"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to the TTCN-3 type definition:

```
type integer Multiple_of_4 (1..4,6..10)
with {
    variant "name as multiple-of-4";
}
```

EXAMPLE 5: The following represents a user-defined top-level simple type definition that is a restriction (with a minLength and a maxLength facet) of another simple type definition, derived by restriction from xsd:string with the addition of an enumeration facet:

```
<xsd:simpleType name="colour">
    <xsd:restriction>
        <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                <xsd:enumeration value="white"/>
                <xsd:enumeration value="black"/>
                <xsd:enumeration value="red"/>
            </xsd:restriction>
        </xsd:simpleType>
        <xsd:minLength value="2"/>
        <xsd:maxLength value="4"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to the TTCN-3 type definition:

```
type enumerated Color { red }
with {
    variant "name as uncapitalized";
}
```

6.1.6 WhiteSpace

The *whiteSpace* facet has no corresponding feature in TTCN-3 but shall be preserved using the whitespace encoding instruction.

EXAMPLE:

```
<xsd:simpleType name="e8">
    <xsd:restriction base="xsd:string">
        <xsd:whiteSpace value="replace"/>
    </xsd:restriction>
</xsd:simpleType>
```

This can be mapped into a charstring, sending information about the whiteSpace facet to the codec.

```
type XSD.String E8
with {
    variant "whiteSpace replace";
    variant "name as uncapitalized";
}
```

For most built-in types the value of the whiteSpace facet shall be set to "*collapse*" and only for the string types *normalizedString* (see clause 6.2.2), *token* (see clause 6.2.2), *language* (see clause 6.2.13), *Name* (see clause 6.2.4) and *NCName* (see clause 6.2.6) are allowed to specify this facet.

6.1.7 MinInclusive

The *minInclusive* XSD facet is only applicable to the numerical types (*integer*, *decimal*, *float*, *double* and their derivatives) and date-time types (*duration*, *dateTime*, *time*, *gYearMonth*, *gYear*, *gMonthDay*, *gDay* and *gMonth*). It specifies the lowest bound of the type's value space, including the bound. This facet is mapped to TTCN-3 depending on the base type of the facet's parent *restriction* element and the value of the facet:

a) if the *minInclusive* facet is applied to a *float* or *double* type (including their derivatives) and its value is one of the special values INF (positive infinity) or NaN (not-a-number), it shall be translated to a list subtyping with the single TTCN-3 value **`infinity`** or **`not_a_number`**, respectively (independent of the value of a *maxInclusive* or *maxEclusive* facet applied to the same type, if any);

b) otherwise, if the *minInclusive* facet is applied to a numeric type, it shall be translated to an inclusive lower bound of a range restriction in TTCN-3. The upper bound of the base type range shall be:

- defined by a *maxInclusive* (see clause 6.1.8) or a *maxEclusive* (see clause 6.1.10) facet, which is a child of the same *restriction* element, if any;
- or inherited from the base type; in case the base type is one of the XSD built-in types *integer*, *decimal*, *float*, *double*, *nonNegativeInteger* or *positiveInteger*, it shall be set to **`infinity`** if not set) (in case of other built-in numerical types the upper bounds of their value spaces are defined in [9]);

c) for the date-time types the facet shall be ignored.

NOTE: The upper bound of the value space of the XSD *float* type is **`3.402823466385288598117041834845 2E38`** ((2^24-1)*2^104) and of the XSD *double* type is **`1.826877046663628444930510004378 6E47`** ((2^53-1)*2^970). However, TTCN-3 does not place the requirement to support these values by TTCN-3 tools. Therefore, to maintain the portability of the generated TTCN-3 code, the upper bound is set to **`infinity`**, if no *maxInclusive* or *maxEclusive* facet is applied. However, users should respect the values above, otherwise the result of producing encoded XML values in undeterministic.

EXAMPLE 1: Mapping of an integer element with a *minInclusive* facet:

```
<xsd:simpleType name="e9a">
    <xsd:restriction base="xsd:integer">
        <xsd:minInclusive value="-5"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to TTCN-3 e.g. as:

```
type XSD.Integer E9a (-5 .. infinity)
with {
    variant "name as uncapitalized";
}
```

EXAMPLE 2: Mapping of a float element with a numeric *minInclusive* value:

```
<xsd:simpleType name="e9b">
    <xsd:restriction base="xsd:float">
        <xsd:minInclusive value="-5"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to TTCN-3 e.g. as:

```
type XSD.Float E9b (-5.0 .. infinity)
with {
    variant "name as uncapitalized";
}
```

EXAMPLE 3: Mapping of a float element with special *minInclusive* values:

```
<xsd:simpleType name="e9c">
    <xsd:restriction base="xsd:float">
        <xsd:minInclusive value="-INF"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to TTCN-3 e.g. as:

```
type XSD.Float E9c (-infinity .. infinity)
with {
    variant "name as uncapitalized";
}

<xsd:simpleType name="e9d">
    <xsd:restriction base="xsd:float">
        <xsd:minInclusive value="INF"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to TTCN-3 e.g. as:

```
type XSD.Float E9d ( infinity )
with {
    variant "name as uncapitalized";
}
```

```
<xsd:simpleType name="e9e">
    <restriction base="xsd:float">
        <xsd:minInclusive value="NaN"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to TTCN-3 e.g. as:

```
type XSD.Float E9e ( not_a_number )
with {
    variant "name as uncapitalized";
}
```

6.1.8 MaxInclusive

The *maxInclusive* facet is only applicable to the numerical types (integer, decimal, float, double and their derivatives) and date-time types (*duration, dateTime*, *time*, *gYearMonth*, *gYear*, *gMonthDay*, *gDay* and *gMonth*). It specifies the upmost bound of the type's value space, including the bound. This facet is mapped to TTCN-3 depending on the base type defined in the facet's parent *restriction* element and the value of the facet:

a) if the *maxInclusive* facet is applied to a *float* or *double* type (including their derivatives) and its value is one of the special values -INF (negative infinity) or NaN (not-a-number), it shall be translated to a list subtyping with the single TTCN-3 value **`-infinity`** or **`not_a_number`**, respectively (independent of the value of a *minInclusive* or *minEclusive* facet applied to the same *restriction* element, if any);

b) otherwise, if the *maxInclusive* facet is applied to a numeric type, it shall be translated to an inclusive upper bound of a range restriction in TTCN-3. The lower bound of the range shall be:

   - defined by a *minInclusive* (see clause 6.1.7) or a min*Eclusive* (see clause 6.1.9) facet, which is a child of the same *restriction* element, if any;

   - or inherited from the base type; in case the base type is one of the XSD built-in types *integer*, *decimal*, *float*, *double*, *nonPositiveInteger* or *negativeInteger*, it shall be set to (**`-infinity`** if not set) (in case of other built-in numerical types the lower bounds of their value spaces are given in [9]);

c) for the date-time types the facet shall be ignored.

NOTE: Note, that the lower bound of the value space of the XSD *float* type is **`-3.40282346638528859811704183484526E38`** ((2^24-1)*2^104) and of the XSD *double* type is **`-1.82687704666362844493051000437864E47`** ((2^53-1)*2^970). However, TTCN-3 does not place the requirement to support these values by TTCN-3 tools. Therefore, to maintain the portability of the generated TTCN-3 code, the lower bound is set to **`-infinity`**, if no *minInclusive* or *minEclusive* facet is applied. However, users should respect the values above, otherwise the result of producing encoded XML values in undeterministic.

EXAMPLE 1: Mapping of elements of type integer with *maxInclusive* facet:

```
<xsd:simpleType name="e10a">
    <xsd:restriction base="xsd:positiveInteger">
        <xsd:maxInclusive value="100"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to TTCN-3 e.g. as:

```
type XSD.PositiveInteger E10a (1 .. 100)
with {
    variant "name as uncapitalized";
}
```

EXAMPLE 2: Mapping of a float type with a numeric *maxInclusive* facet:

```
<xsd:simpleType name="e10b">
    <xsd:restriction base="xsd:float">
        <xsd:maxInclusive value="-5"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to TTCN-3 e.g. as:

```
type XSD.Float E10b ( -infinity .. -5.0 )
//pls. note that XSD allows an integer-like value notation for float types but TTCN-3 does not!
with {
    variant "name as uncapitalized";
}
```

EXAMPLE 3: Mapping of a float type with specific-value *maxInclusive* facets:

```
<xsd:simpleType name="e10c">
    <xsd:restriction base="xsd:float">
        <xsd:maxInclusive value="INF"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to TTCN-3 e.g. as:

```
type XSD.Float E10c (-infinity .. infinity)
    with {
        variant "name as uncapitalized";
    }

    <xsd:simpleType name="e10d">
        <xsd:restriction base="float">
            <maxInclusive value="NaN"/>
        </xsd:restriction>
    </xsd:simpleType>
```

Is mapped to TTCN-3 e.g. as:

```
type XSD.Float E10d ( not_a_number )
with {
    variant "name as uncapitalized";
}
```

6.1.9 MinExclusive

The XSD facet *minExclusive* is similar to *minInclusive* but the specified bound is not part of the range. It is also applicable to the XSD numerical and date-time types (see clause 6.1.7). This facet is mapped to TTCN-3 depending on the base type defined in the facet's parent *restriction* element and the value of the facet:

a) if the *minExclusive* facet is applied to a *float* or *double* type and its value is one of the special values INF (positive infinity) or NaN (not-a-number), this type shall not be translated to TTCN-3;

NOTE 1: If the value of the *minExclusive* facet is INF or NaN, this result an empty type in XSD, but empty types do not exist in TTCN-3.

b) otherwise, if the *minExclusive* facet is applied to an *integer*, *float*, *double* or *decimal* type, it shall be translated to an exclusive lower bound of a range restriction in TTCN-3; the value of the bound shall be the value of the *minExclusive* facet;

c) in case b) the upper bound of the range shall be:

- defined by a *maxInclusive* (see clause 6.1.8) or a *maxEclusive* (see clause 6.1.10) facet, which is a child of the same *restriction* element, if any;
- or inherited from the base type; in case the base type is one of the XSD built-in types *integer*, *decimal*, *float*, *double*, *nonNegativeInteger* or *positiveInteger*, it shall be set to **`infinity`** (in case of other built-in numerical types the upper bounds of their value spaces are defined in [9]);

d) for the date-time types the facet shall be ignored.

NOTE 2: The upper bound of the value space of the XSD *float* type is **3.402823466385288598117041834845 2E38** ((2^24-1)*2^104) and of the XSD *double* type is **1.826877046663628444930510004378 6E47** ((2^53-1)*2^970). However, TTCN-3 does not place the requirement to support these values by TTCN-3 tools. Therefore, to maintain the portability of the generated TTCN-3 code, the upper bound is set to **infinity**, if no *maxInclusive* or *maxEclusive* facet is applied. However, users should respect the values above, otherwise the result of producing encoded XML values in undeterministic.

EXAMPLE 1: Mapping of the *minExclusive* facet applied to an *integer* type:

```
<xsd:simpleType name="e11a">
    <xsd:restriction base="xsd:integer">
        <xsd:minExclusive value="-5"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to TTCN-3 e.g. as:

```
type XSD.Integer E11a (!-5 .. infinity)
with {
    variant "name as uncapitalized";
}
```

EXAMPLE 2: Mapping of a *float* type with *minExclusive* facet:

```
<xsd:simpleType name="e11b">
    <xsd:restriction base="xsd:float">
        <xsd:minExclusive value="-5"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to TTCN-3 e.g. as:

```
type XSD.Float E11b (!-5.0 .. infinity)
//pls. note that XSD allows an integer-like value notation for float types but TTCN-3 does not!
with {
    variant "name as uncapitalized";
}

<xsd:simpleType name="e11c">
    <xsd:restriction base="ns:e10b">
        <xsd:minExclusive value="-6"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped to TTCN-3 e.g. as:

```
type XSD.Float E11c (!-6.0 .. -5.0)
with {
    variant "name as uncapitalized";
}

<xsd:simpleType name="e11d">
    <xsd:restriction base="xsd:float">
        <xsd:minExclusive value="INF"/>
    </xsd:restriction>
</xsd:simpleType>

// No corresponding TTCN-3 type is produced
```

6.1.10 MaxExclusive

The XSD facet *maxExclusive* is similar to *maxInclusive* but the specified bound is not part of the range. It is also applicable to the XSD numerical and date-time types (see clause 6.1.8). This facet is mapped to TTCN-3 depending on the base type defined in the facet's parent *restriction* element and the value of the facet:

a) if the *maxExclusive* facet is applied to a *float* or *double* type and its value is one of the special values -INF (negative infinity) or NaN (not-a-number), this type shall not be translated to TTCN-3;

NOTE 1: If the value of the *maxExclusive* facet is -INF or NaN, this result an empty type in XSD, but empty types do not exist in TTCN-3.

b) otherwise, if the *maxExclusive* facet is applied to an *integer*, *float*, *double* or *decimal* type, it shall be translated to an exclusive upper bound of a range restriction in TTCN-3; the value of the bound shall be the value of the *maxExclusive* facet;

c) in case b) the lower bound of the range shall be:

- defined by a *minInclusive* (see clause 6.1.7) or a min*Eclusive* (see clause 6.1.9) facet, which is a child of the same *restriction* element, if any;
- or inherited from the base type; in case the base type is one of the XSD built-in types *integer*, *decimal*, *float*, *double*, *nonPositiveInteger* or *negativeInteger*, it shall be set to **`-infinity`** (in case of other built-in numerical types the lower bounds of their value spaces are given in [9]);

d) for the date-time types the facet shall be ignored.

NOTE 2: The lower bound of the value space of the XSD *float* type is **`-3.40282346638528859811704183484526E38`** ((2^24-1)*2^104) and of the XSD *double* type is **`-1.82687704666362844493051000437864E47`** ((2^53-1)*2^970). However, TTCN-3 does not place the requirement to support these values by TTCN-3 tools. Therefore, to maintain the portability of the generated TTCN-3 code, the lower bound is set to **`-infinity`**, if no *minInclusive* or *minEclusive* facet is applied. However, users should respect the values above, otherwise the result of producing encoded XML values in undeterministic.

EXAMPLE 1: Mapping of a *maxExclusive* facet applied to a type, which is derivative of *integer*:

```
<xsd:simpleType name="e12a">
    <xsd:restriction base="xsd:positiveInteger">
        <xsd:maxExclusive value="100"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped in TTCN-3 e.g. to:

```
type XSD.PositiveInteger E12a (1 .. !100)
with {
    variant "name as uncapitalized";
}
```

EXAMPLE 2: Mapping of a *maxExclusive* facet applied to the *float* type:

```
<xsd:simpleType name="e12b">
    <xsd:restriction base="xsd:float">
        <xsd:maxExclusive value="-5"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped in TTCN-3 e.g. as:

```
type XSD.Float E12b (-infinity .. ! -5.0)
//pls. note that XSD allows an integer-like value notation for float types but TTCN-3 does not!
with {
    variant "name as uncapitalized";
}

<xsd:simpleType name="e12c">
    <xsd:restriction base="ns:e9b">
        <xsd:maxExclusive value="-4"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is mapped in TTCN-3 e.g. as:

```
type XSD.Float E12c (-5.0 .. ! -4.0)
with {
    variant "name as uncapitalized";
}

<xsd:simpleType name="e12d">
    <xsd:restriction base="xsd:float">
        <xsd:maxExclusive value="-INF"/>
    </xsd:restriction>
</xsd:simpleType>
```

```
// No corresponding TTCN-3 type is produced.
```

6.1.11 Total digits

This facet defines the value range allowed for numeric values. In case of integer values it equals the maximum number of digits a numeric value is allowed to have. In case of the XSD decimal values it limits the value range, but not the number of leading zero digits and trailing fractional zero digits.

It shall be mapped to TTCN-3 using ranges by converting the value of *totalDigits* to the proper boundaries of the numeric type in question.

EXAMPLES:

```
<xsd:simpleType name="e13">
    <xsd:restriction base="xsd:negativeInteger">
        <xsd:totalDigits value="3"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is translated to TTCN-3 e.g. as:

```
type XSD.NegativeInteger E13 (-999 .. -1)
with {
    variant "name as uncapitalized";
}
```

```
<xsd:simpleType name='restrictedDecimal'>
  <xsd:restriction base='xsd:decimal'>
    <xsd:totalDigits value='4'/>
  </xsd:restriction>
</xsd:simpleType>
```

Is translated to TTCN-3 e.g. as:

```
type XSD.Decimal RestrictedDecimal (-9999.0 .. 9999.0)
with {
    variant "name as uncapitalized";
}
```

6.1.12 Fraction digits

This facet defines the maximum number of digits of XML decimal values, following the decimal point (the period). TTCN-3 does not have similar feature to constrain the format of float values. This facet shall be mapped to the "fractionDigits" encoding instruction attached to the generated TTCN-3 definition.

EXAMPLE:

```
<xsd:simpleType name='celsiusBodyTemp'>
  <xsd:restriction base='xsd:decimal'>
    <xsd:totalDigits value='4'/>
    <xsd:fractionDigits value='1'/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:element name='actualTemp' type='ns:celsiusBodyTemp'/>
```

Is translated to TTCN-3 e.g. as:

```
type XSD.Decimal CelsiusBodyTemp (-9999.0 .. 9999.0)
with {
    variant "name as uncapitalized";
    variant "fractionDigits 1";
}

type CelsiusBodyTemp ActualTemp
with {
    variant "name as uncapitalized";
    variant "element";
}
```

```
//Thus
const ActualTemp c_temp1:= 9999.0
// is allowed and should be encoded as e.g.
<actualTemp>9999</actualTemp>
//or as
<actualTemp>09999.0</actualTemp>
//And
const ActualTemp c_temp2:= 99.99
// is allowed and should be encoded as e.g.
<actualTemp>99.9</actualTemp>
//or as
<actualTemp>0099.9</actualTemp>

//And
const ActualTemp c_temp3:= 9.999E-1
// is allowed and should be encoded as e.g.
<actualTemp>99.9</actualTemp>
//or as
<actualTemp>0099.9</actualTemp>
```

6.1.13 Not specifically mapped facets

Whenever an XSD facet element is not mapped to a TTCN-3 by one of the preceding clauses, it shall be mapped to a "transparent …" encoding instruction containing the name and the value of the XSD facet element.

The content of the encoding instruction shall be of the form **`transparent`** `<facet> '<value>'` where `<facet>` is the XSD facet element's name and `<value>` is the content of the value attribute of that facet element.

NOTE: Since the **pattern** and enumeration facets are the only facets which can contain the " character and this is only possible for XSD **string** based types which will be mapped to value or pattern subtype restrictions (see clauses 5, 5.5 and 6), it is never necessary to quote the " character in any valid pattern value.

EXAMPLE:

```
<xsd:simpleType name="decimalWithWhole">
    <xsd:restriction base="xsd:decimal">
        <xsd:pattern value="[0-9][.][0-9]*"/>
    </xsd:restriction>
</xsd:simpleType>
```

Is translated to TTCN-3 e.g. as:

```
type XSD.Decimal DecimalWithWhole
with {
    variant "name as uncapitalized";
    variant "transparent pattern '[0-9][.][0-9]*'";
}
```

6.2 String types

6.2.0 General

XSD string types shall generally be converted to TTCN-3 as subtypes of *universal charstring* or *octetstring* as specified in this and in subsequent clauses. For an overview of the allowed facets please refer to table 2. The following clauses specify the mapping of all string types of XSD.

To support mapping, the following type definitions are added to the built-in data types (utf8string is declared as a UTF-8 encoded subtype of universal charstring in clause E.2.2.0 of ETSI ES 201 873-1 [1]):

```
type utf8string XMLCompatibleString
 (
  char(0,0,0,9).. char(0,0,0,9),
  char(0,0,0,10)..char(0,0,0,10),
  char(0,0,0,13)..char(0,0,0,13),
  char(0,0,0,32)..char(0,0,215,255),
  char(0,0,224,0)..char(0,0,255,253),
  char(0,1,0,0)..char(0,16,255,253)
);
```

```
type utf8string XMLStringWithNoWhitespace
 (
  char(0,0,0,33)..char(0,0,215,255),
  char(0,0,224,0)..char(0,0,255,253),
  char(0,1,0,0)..char(0,16,255,253)
);

type utf8string XMLStringWithNoCRLFHT
 (
  char(0,0,0,32)..char(0,0,215,255),
  char(0,0,224,0)..char(0,0,255,253),
  char(0,1,0,0)..char(0,16,255,253)
);
```

6.2.1 String

The *string* type shall be translated to TTCN-3 as an XML compatible restriction of the universal charstring:

```
type XSD.XMLCompatibleString String
with {
    variant "XSD:string";
}
```

6.2.2 Normalized string

The *normalizedString* type shall be translated to TTCN-3 using the following XML compatible restricted subtype of the universal charstring:

```
type XSD.XMLStringWithNoCRLFHT NormalizedString
with {
     variant "XSD:normalizedString";
}
```

6.2.3 Token

The *token* type shall be translated to TTCN-3 using the built-in data type NormalizedString:

```
type XSD.NormalizedString Token
with {
    variant "XSD:token";
}
```

6.2.4 Name

The *Name* type shall be translated to TTCN-3 using the following XML compatible restricted subtype of the universal charstring:

```
type XSD.XMLStringWithNoWhitespace Name
with {
    variant "XSD:Name";
}
```

6.2.5 NMTOKEN

The *NMTOKEN* type shall be translated to TTCN-3 using the following XML compatible restricted subtype of the universal charstring:

```
type XSD.XMLStringWithNoWhitespace NMTOKEN
with {
    variant "XSD:NMTOKEN";
}
```

6.2.6 NCName

The *NCName* type shall be translated to TTCN-3 using the built-in data type Name:

```
type XSD.Name NCName
with {
    variant "XSD:NCName";
}
```

6.2.7 ID

The *ID* type shall be translated to TTCN-3 using the built-in data type NCName:

```
type XSD.NCName ID
with {
    variant "XSD:ID";
}
```

6.2.8 IDREF

The *IDREF* type shall be translated to TTCN-3 using the built-in data type NCName:

```
type XSD.NCName IDREF
with {
    variant "XSD:IDREF";
}
```

6.2.9 ENTITY

The *ENTITY* type shall be translated to TTCN-3 using the built-in data type NCName:

```
type XSD.NCName ENTITY
with {
    variant "XSD:ENTITY";
};
```

6.2.10 Hexadecimal binary

The *hexBinary* type shall be translated to TTCN-3 using a plain octetstring:

```
type octetstring HexBinary
with {
    variant "XSD:hexBinary";
}
```

No pattern shall be specified for *hexBinary* types.

6.2.11 Base 64 binary

The XSD *base64Binary* type shall be translated to an octetstring in TTCN-3. When encoding elements of this type, the XML codec will invoke automatically an appropriate base64 encoder; when decoding XML instance content, the base64 decoder will be called.

The base64Binary type shall be mapped to the TTCN-3 type:

```
type octetstring Base64Binary
with {
    variant "XSD:base64Binary";
}
```

EXAMPLE:

```
<xsd:element name="E14 type="base64Binary"/>
```

Is translated TTCN-3 e.g. as as:

```
type XSD.Base64Binary E14;
```

and the template:

```
template E14 MyBase64BinaryTemplate := char2oct("TTCN-3")
```

Can be encoded e.g. as:

```
<E14>VFRDTi0z</E14>
```

6.2.12 Any URI

The *anyURI* type shall be translated to TTCN-3 as an XML compatible restricted subtype of the universal charstring:

```
type XSD.XMLStringWithNoCRLFHT AnyURI
with {
    variant "XSD:anyURI";
}
```

6.2.13 Language

The *language* type shall be translated to the TTCN-3 type:

```
type charstring Language (pattern "[a-zA-Z]#(1,8)(-\w#(1,8))#(0,)")
with {
    variant "XSD:language";
}
```

6.2.14 NOTATION

The XSD *NOTATION* type shall not be translated to TTCN-3.

6.3 Integer types

6.3.0 General

XSD integer types shall generally be converted to TTCN-3 as subtypes of integer-based types. For an overview of the allowed facets please refer to table 2. Clauses 6.3.1 to 6.3.13 specify the mapping of all integer types of XSD.

6.3.1 Integer

The *integer* type is not range-restricted in XSD and shall be translated to TTCN-3 as a plain *integer*:

```
type integer Integer
with {
    variant "XSD:integer";
}
```

6.3.2 Positive integer

The *positiveInteger* type shall be translated to TTCN-3 as the range-restricted *integer*:

```
type integer  PositiveInteger (1 .. infinity)
  with { variant "XSD:positiveInteger"};
```

6.3.3 Non-positive integer

The *nonPositiveInteger* type shall be translated to TTCN-3 as the range-restricted *integer*:

```
type integer NonPositiveInteger (-infinity .. 0)
with {
    variant "XSD:nonPositiveInteger";
}
```

6.3.4 Negative integer

The *negativeInteger* type shall be translated to TTCN-3 as the range-restricted *integer*:

```
type integer  NegativeInteger (-infinity .. -1)
with {
    variant "XSD:negativeInteger";
};
```

6.3.5 Non-negative integer

The *nonNegativeInteger* type shall be translated to TTCN-3 as the range-restricted *integer*:

```
type integer NonNegativeInteger (0 .. infinity)
with {
    variant "XSD:nonNegativeInteger";
}
```

6.3.6 Long

The *long* type is 64bit based in XSD and shall be translated to TTCN-3 as a plain *longlong* as defined in clause E.2.1.3 of ETSI ES 201 873-1 [1]:

```
type longlong Long
with {
    variant "XSD:long";
}
```

6.3.7 Unsigned long

The *unsignedLong* type is 64bit based in XSD and shall be translated to TTCN-3 as a plain *unsignedlonglong* as defined in clause E.2.1.3 of ETSI ES 201 873-1 [1]:

```
type unsignedlonglong UnsignedLong
with {
    variant "XSD:unsignedLong";
}
```

6.3.8 Int

The *int* type is 32bit based in XSD and shall be translated to TTCN-3 as a plain *long* as defined in clause E.2.1.2 of ETSI ES 201 873-1 [1]):

```
type long Int
with {
    variant "XSD:int";
}
```

6.3.9 Unsigned int

The *unsignedInt* type is 32bit based in XSD and shall be translated to TTCN-3 as a plain *unsignedlong* as defined in clause E.2.1.2 of ETSI ES 201 873-1 [1]:

```
type unsignedlong UnsignedInt
with {
    variant "XSD:unsignedInt";
}
```

6.3.10 Short

The *short* type is 16bit based in XSD and shall be translated to TTCN-3 as a plain *short* as defined in clause E.2.1.1 of ETSI ES 201 873-1 [1]:

```
type short Short
with {
    variant "XSD:short";
}
```

6.3.11 Unsigned Short

The *unsignedShort* type is 16bit based in XSD and shall be translated to TTCN-3 as a plain *unsignedshort* as defined in clause E.2.1.1 of ETSI ES 201 873-1 [1]:

```
type unsignedshort UnsignedShort
with {
    variant "XSD:unsignedShort";
}
```

6.3.12 Byte

The *byte* type is 8bit based in XSD and shall be translated to TTCN-3 as a plain *byte* as defined in clause E.2.1.0 of ETSI ES 201 873-1 [1]:

```
type byte Byte
with {
    variant "XSD:byte";
}
```

6.3.13 Unsigned byte

The *unsignedByte* type is 8bit based in XSD and shall be translated to TTCN-3 as a plain *unsignedbyte* as defined in clause E.2.1.0 of ETSI ES 201 873-1 [1]:

```
type unsignedbyte UnsignedByte
with {
    variant "XSD:unsignedByte";
}
```

6.4 Float types

6.4.0 General

XSD float types are generally converted to TTCN-3 as subtypes of *float*. For an overview of the allowed facets refer to table 2 in clause 6.1. The following clauses specify the mapping of all float types of XSD.

6.4.1 Decimal

The *decimal* type shall be translated to TTCN-3 as a plain *float*:

```
type float Decimal (!-infinity .. !infinity)
with {
    variant "XSD:decimal";
}
```

6.4.2 Float

The *float* type shall be translated to TTCN-3 as an *IEEE754float* as defined in clause E.2.1.4 of ETSI ES 201 873-1 [1]:

```
type IEEE754float Float
with {
    variant "XSD:float";
}
```

6.4.3 Double

The *double* type shall be translated to TTCN-3 as an *IEEE754double* as defined in clause E.2.1.4 of ETSI ES 201 873-1 [1]:

```
type IEEE754double Double
with {
    variant "XSD:double";
}
```

6.5 Time types

6.5.0 General

XSD time types shall generally be converted to TTCN-3 as pattern restricted subtypes of *charstring*. For an overview of the allowed facets refer to table 2. Details on the mapping of all time types of XSD are given in the following.

For the definition of XSD time types, the supplementary definitions below are used. These definitions are part of the module XSD (see annex A). As a consequence, in case of both implicit and explicit mappings, it shall be possible to use their identifiers in other (user defined) modules but also, it shall be possible to reference these definitions by using their qualified names (e.g. XSD.year).

```
const charstring
  dash := "-",
  cln  := ":",
  year := "[0-9]#(4)",
  yearExpansion := "-#(,1)([1-9][0-9]#(0,))#(,1)",
  month := "(0[1-9]|1[0-2])",
  dayOfMonth := "(0[1-9]|[12][0-9]|3[01])",
  hour := "([01][0-9]|2[0-3])",
  minute := "([0-5][0-9])",
  second := "([0-5][0-9]|60)",
  sFraction := "(.[0-9]#(1,))#(,1)",
  endOfDayExt := "24:00:00(.0#(1,))#(,1)",
  nums := "[0-9]#(1,)",
  ZorTimeZoneExt := "(Z|[\+\-]((0[0-9]|1[0-3]):[0-5][0-9]|14:00))#(,1)",
  durTime := "(T[0-9]#(1,)"&
             "(H([0-9]#(1,)(M([0-9]#(1,)(S|.[0-9]#(1,)S))#(,1)|.[0-9]#(1,)S|S))#(,1)|" &
             "M([0-9]#(1,)(S|.[0-9]#(1,)S)|.[0-9]#(1,)M)#(,1)|"&
             "S|"&
             ".[0-9]#(1,)S))"
```

NOTE 1: The patterns above implement the syntactical restrictions of ISO 8601 [i.2] and XSD (e.g.month 00 or 13, day 00 or 32 are disallowed) but the semantical restrictions of XSD (e.g. 2001-02-29 is a non existing date as 2001 is not a leap year) are not imposed.

NOTE 2: Please note that XSD version 1.0 referred to ISO 8601 [i.2] for date representations, which disallowed year 0000, while XSD version 1.1 allows year 0000 (see https://www.w3.org/TR/xmlschema11-2/#dateTime [i.14]).

NOTE 3: The patterns in the subsequent clauses, i.e. the text between the double quotes, need to be one continuous string without whitespace when being used in a TTCN-3 code. The lines below are cut for pure editorial reasons, to fit the text to the standard page size of the present document.

NOTE 4: The second 60 value is only used to denote a leap second.

6.5.1 Duration

The *duration* type shall be translated to TTCN-3 using the following pattern-restricted charstring:

```
type charstring  Duration (pattern
.."{dash}#(,1)P({nums}(Y({nums}(M({nums}D{durTime}#(,1)|{durTime}#(,1))|D{durTime}#(,1))|" &
  "{durTime}#(,1))|M({nums}D{durTime}#(,1)|{durTime}#(,1))|D{durTime}#(,1))|{durTime})"
  )
with {
    variant "XSD:duration";
}
```

6.5.2 Date and time

The *dateTime* type shall be translated to TTCN-3 using the following pattern-restricted charstring:

```
type charstring  DateTime (pattern
  "{yearExpansion}{year}{dash}{month}{dash}{dayOfMonth}T({hour}{cln}{minute}{cln}{second}" &
  "{sFraction}|{endOfDayExt}){ZorTimeZoneExt}"
  )
with {
    variant "XSD:dateTime";
}
```

6.5.3 Time

The *time* type shall be translated to TTCN-3 using the following pattern-restricted charstring:

```
type charstring  Time (pattern
  "({hour}{cln}{minute}{cln}{second}{sFraction}|{endOfDayExt}){ZorTimeZoneExt}"
  )
with {
    variant "XSD:time";
}
```

6.5.4 Date

The *date* type shall be translated to TTCN-3 using the following pattern-restricted charstring:

```
type charstring  Date (pattern
  "{yearExpansion}{year}{dash}{month}{dash}{dayOfMonth}{ZorTimeZoneExt}"
  )
with {
    variant "XSD:date";
}
```

6.5.5 Gregorian year and month

The *gYearMonth* type shall be translated to TTCN-3 using the following pattern-restricted charstring:

```
type charstring  GYearMonth (pattern
  "{yearExpansion}{year}{dash}{month}{ZorTimeZoneExt}"
  )
with {
    variant "XSD:gYearMonth";
}
```

6.5.6 Gregorian year

The *gYear* type shall be translated to TTCN-3 using the following pattern-restricted charstring:

```
type charstring  GYear (pattern
  "{yearExpansion}{year}{ZorTimeZoneExt}"
  )
with {
    variant "XSD:gYear";
}
```

6.5.7 Gregorian month and day

The *gMonthDay* type shall be translated to TTCN-3 using the following pattern-restricted charstring:

```
type charstring  GMonthDay (pattern
  "{dash}{dash}{month}{dash}{dayOfMonth}{ZorTimeZoneExt}"
  )
with {
    variant "XSD:gMonthDay";
}
```

6.5.8 Gregorian day

The *gDay* type shall be translated to TTCN-3 using the following pattern-restricted charstring:

```
type charstring  GDay (pattern
  "{dash}{dash}{dash}{dayOfMonth}{ZorTimeZoneExt}"
  )
with {
    variant "XSD:gDay";
}
```

6.5.9 Gregorian month

The *gMonth* type shall be translated to TTCN-3 using the following pattern-restricted charstring:

```
type charstring  GMonth (pattern
  "{dash}{dash}{month}{ZorTimeZoneExt}"
  )
with {
    variant "XSD:gMonth";
}
```

6.6 Sequence types

6.6.0 General

XSD sequence types shall generally be converted to TTCN-3 as a *record of* their respective base types. For an overview of the allowed facets refer to table 2. The following clauses specify the mapping of all sequence types of XSD.

6.6.1 NMTOKENS

The XSD *NMTOKENS* type shall be mapped to TTCN-3 using a *record of* construct of type *NMTOKEN*:

```
type record of XSD.NMTOKEN NMTOKENS
with {
    variant "XSD:NMTOKENS";
}
```

6.6.2 IDREFS

The XSD *IDREFS* type shall be mapped to TTCN-3 using a *record of* construct of type *IDREF*:

```
type  record of IDREF IDREFS
with {
     variant "XSD:IDREFS";
}
```

6.6.3 ENTITIES

The XSD *ENTITIES* type shall be mapped to TTCN-3 using a *record of* construct of type *ENTITY*:

```
type record of ENTITY ENTITIES
with {
    variant "XSD:ENTITIES";
}
```

6.6.4 QName

The XSD *QName* type shall be translated to the TTCN-3 type QName as given below:

```
type record QName {
    AnyURI uri optional,
    NCName name
}
with {
    variant "XSD:QName";
}
```

When encoding an element of type QName (or derived from QName), if the encoder detects the presence of an URI and this is different from the target namespace, the following encoding shall result (the assumed target namespace is http://www.example.org/).

EXAMPLE:

```
<xsd:element name="e14a">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="name" type="xsd:QName"/>
            <xsd:element name="refId" type="xsd:integer"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Will be translated to TTCN-3 e.g. as:

```
type record E14a
{
    QName name,
    integer refId
}
with {
    variant "name as uncapitalized";
    variant "element";
}
```

Which allows e.g. the following encoding:

```
template E14a t_E14a:=
{
    name:={
        uri:="http://www.organization.org/",
        name:="someName"
},
    refId:=10
}

<?xml version="1.0" encoding="UTF-8"?>
<E14a xmlns="http://www.example.org/">
<name xmlns:ns="http://www.organization.org/">ns:someName</name>
```

```
    <refId>10</refId>
    </E14a>
```

6.7 Boolean type

The XSD *boolean* type shall be mapped to the TTCN-3 *boolean* type:

```
type boolean Boolean
with {
     variant "XSD:boolean";
}
```

During translation of XSD *boolean* values it is necessary to handle all four encodings that XSD allows for Booleans ("true", "false", "0", and "1"); this shall be realized by using the "text" encoding instruction:

```
type XSD.Boolean MyBooleanType
with {
  variant "text 'true' as '1'";
  variant "text 'false' as '0'";
}
```

6.8 AnyType and anySimpleType types

The XSD *anySimpleType* can be considered as the base type of all primitive data types, while the XSD *anyType* is the base type of all complex definitions and the anySimpleType.

The *anySimpleType* shall be translated to the type AnySimpleType or to the type anytype (dependent on some tool configuration beyond the scope of the present document):

```
type XSD.XMLCompatibleString AnySimpleType

with {
   variant "XSD:anySimpleType";
}
```

EXAMPLE 1: Mapping of an element of anySimpleType:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.example.org/AnySimpleType"
xmlns:tns="http://www.example.org/AnySimpleType">

    <xsd:element name="anythingSimple" type="xsd:anySimpleType"></xsd:element>

</xsd:schema>
```

Is translated to the TTCN-3 module e.g. as:

```
module http_www_example_org_AnySimpleType {

  import from XSD all;

  type XSD.AnySimpleType AnythingSimple
  with {
    variant "name as uncapitalized";
    variant "element";
  }
}
with {
    encode "XML";
    variant "namespace as 'http://www.example.org/AnySimpleType' prefix 'tns'";
    variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
    variant "elementFormQualified";
}
```

And the template:

```
template AnythingSimple t_anySimpleType_string := "something goes here";
```

Should be encoded in XML e.g. as:

```
<tns:anythingSimple xmlns:tns='http://www.example.org/AnySimpleType'>something goes
here</tns:anythingSimple>
```

If the *anySimpleType* is translated to the TTCN-3 type **anytype**, values and templates of that type shall only contain type variants which are either the types translated from XSD simple types or types where the root type is one of the TTCN-3 base types integer, float, boolean, charstring , universal charstring, bitstring, hexstring or octetstring, or the type XSD.AnySimpleType.

NOTE 1: In TTCN-3 **anytype**-s only the locally defined and the directly imported data and address types are visible (i.e. the known types); e.g. to use the built-in XSD types the **XSD** module (see annex A of the present document) need explicitly be imported. See more details regarding the use of TTCN-3 **anytype**-s in ETSI ES 201 873-1 [1].

EXAMPLE 2: Mapping of an element of *anySimpleType* to **anytype**:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.example.org/AnySimpleType"
xmlns:tns="http://www.example.org/AnySimpleType">

  <xsd:element name="anythingSimple" type="xsd:anySimpleType"></xsd:element>

</xsd:schema>

// is translated to the TTCN-3 module
module http_www_example_org_AnySimpleType {

  import from XSD all;

  type anytype AnythingSimple
  with {
    variant "name as uncapitalized";
    variant "element";
  }
}
with {
encode "XML";
variant "namespace as 'http://www.example.org/AnySimpleType' prefix 'tns'";
variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
variant "elementFormQualified";
}

// And the template
template AnythingSimple t_anySimpleType_string := { AnySimpleType := "something goes here" };
```

Should be encoded in XML e.g. as

```
<tns:anythingSimple xmlns:tns='http://www.example.org/AnySimpleType'>something goes
here</tns:anythingSimple>

// suitable templates
template AnythingSimple t_anySimpleType_integer := { integer := 25 };
template AnythingSimple t_anySimpleType_float   := { float := 25.9 };
template AnythingSimple t_anySimpleType_boolean := { boolean := true };
```

The XSD *anyType* shall be translated to the TTCN-3 type AnyType or AnyComplexType (dependent on some tool configuration beyond the scope of the present document):

```
type record AnyType {
    record of XSD.String embed_values optional,
    record length (1 .. infinity) of XSD.String attr optional,
    record of XSD.String elem_list
}
with {
    variant "XSD:anyType";
    variant "embedValues";
    variant(attr) "anyAttributes";
    variant(elem_list) "anyElement";
}
//or
```

```
    type record AnyComplexType {
    record of XSD.String embed_values optional,
    record length (1 .. infinity) of anytype attr optional,
    record of anytype elem_list
}
with {
    variant "XSD:anyType";
    variant "embedValues";
    variant(attr) "anyAttributes";
    variant(elem_list) "anyElement";
}
```

For values or templates of the type AnyComplexType, elements of the field **`attr`** shall contain only type variants translated from XSD *attribute* definitions or the type XSD.String while elements of the field **`elem_list`** shall contain only type variants translated from XSD *element* definitions or the type XSD.String.

An element of *anyType* is able to carry any syntactically valid (well-formed) XML content, including mixed content. Each TTCN-3 element of the field **`attr`** shall contain a complete, valid XML attribute, including its name and value. Each TTCN-3 element of the field **`elem_list`** shall contain a syntactically valid XML element. If the **`embed_values`** field is not omitted in the TTCN-3 value or template instance, its content shall be handled according to clause 7.6.8.

NOTE 2: TTCN-3 values and templates corresponding to simple-type XML elements will omit the **`embed_values`** and **`attr`** fields and will contain a single element in the **`elem_list`** field.

EXAMPLE 3: Mapping of *anyType* element to AnyType:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
    targetNamespace=http://www.example.org/AnyType
    xmlns:tns="http://www.example.org/AnyType">

    <xsd:element name='anything' type='xsd:anyType'/>

</xsd:schema>
```

Is translated to the TTCN-3 module e.g. as:

```
module http_www_example_org_AnyType {

  import from XSD all;

  type XSD.AnyType Anything
  with {
    variant "name as uncapitalized";
    variant "element";
  }
}
with {
  encode "XML";
  variant "namespace as 'http://www.example.org/AnyType' prefix 'tns'";
  variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

And the templates below:

```
template Anything t_simpleElementOnly := {
  embed_values := omit,
  attr := omit,
  elem_list := {"<something>1</something>"}
}
```

Should be encoded in XML e.g. as:

```
<tns:anything xmlns:tns='http://www.example.org/AnyType'>
  <something>1</something>
</tns:anything>


template Anything t_simpleElementOnly_wNS := {
  embed_values := omit,
  attr := omit,
  elem_list := {"<ns:something xmlns:ns='http://www.example.org/other'>1</ns:something>"}
}
```

Should be encoded in XML e.g. as:

```
<tns:anything xmlns: tns='http://www.example.org/AnyType'>
  <ns:something xmlns:ns='http://www.example.org/other'>1</ns:something>
</tns:anything>


template Anything t_attrElement_notMixed := {
  embed_values := omit,
  attr :={"someattr='1'"},
  elem_list := {
    "<ns:a xmlns:ns='http://www.example.org/other'>product</ns:a>",
    "<ns1:b xmlns:ns1='http://www.example.org/other_1'>2</ns1:b>"
  }
}
```

Should be encoded in XML e.g. as:

```
<tns:anything xmlns:tns='http://www.example.org/AnyType' someattr='1'>
  <ns:a xmlns:ns='http://www.example.org/other'>product</ns:a>
  <ns1:b xmlns:ns1='http://www.example.org/other_1'>2</ns1:b>
</tns:anything>


template Anything t_attrElement_Mixed := {
  embed_values := {"The ordered ", " has arrived ", "Wait for further information."},
  attr :={"someattr='1'"},
  elem_list := {
    "<ns:a xmlns:ns='http://www.example.org/other'>product</ns:a>",
    "<ns:b xmlns:ns='http://www.example.org/other_1'>2</ns:b>"
  }
}
```

Should be encoded in XML e.g. as:

```
<tns:anything xmlns:tns='http://www.example.org/AnyType' someattr='1'>The ordered <ns:a
  xmlns:ns='http://www.example.org/other'>product</ns:a> has arrived <ns:b
  xmlns:ns='http://www.example.org/other_1'>2</ns:b>Wait for further information.
</tns:anything>
*/
```

EXAMPLE 4: Mapping of nillable anyType element:

```
<?xml version="1.0" encoding="UTF-8"?>
  <xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
    targetNamespace="http://www.example.org/AnyType"
    xmlns:tns="http://www.example.org/AnyType">

    <xsd:element name='anything-nil' type='xsd:anyType' nillable='true'/>

</xsd:schema>
```

Is translated to the TTCN-3 module e.g. as:

```
module http_www_example_org_AnyType {

  import from XSD all;

  type record Anything_nil
  {
    XSD.AnyType content optional
  }
  with {
    variant "name as 'anything-nil'";
    variant "useNil";
    variant "element";
  }
}
  with {
    encode "XML";
    variant "namespace as 'http://www.example.org/AnyType' prefix 'tns'";
    variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
  }
```

And the template:

```
template Anything_nil t_element_nilled := {
  content := omit
}
```

Should be encoded in XML e.g. as:

```
<tns:anything-nil xmlns:tns='http://www.example.org/AnyType' xsi:nil='true'/>
```

EXAMPLE 5: Mapping of *anyType* element to AnyComplexType:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
  targetNamespace=http://www.example.org/AnyComplexType
  xmlns:tns="http://www.example.org/AnyComplexType">

  <xsd:element name='anything' type='xsd:anyType'/>
  <xsd:element name='something' type='xsd:int'/>
  <xsd:attribute name='someattr' type='xsd:string'/>

</xsd:schema>

// is translated to the TTCN-3 module
module http_www_example_org_AnyComplexType {

  import from XSD all;

  type XSD.AnyComplexType Anything
  with {
    variant "name as uncapitalized";
    variant "element"
  }
  type XSD.Int Something
  with {
    variant "name as uncapitalized";
    variant "element"
  }
  type XSD.String Someattr
  with {
    variant "name as uncapitalized";
    variant "attribute"
  }

}
with {
encode "XML";
variant "namespace as 'http://www.example.org/AnyComplexType' prefix 'tns'";
variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'"
}

// And the templates below
template Anything t_simpleElementOnly := {
  embed_values := omit,
  attr := omit,
  elem_list := {Something:=1}
}
```

Should be encoded in XML e.g. as:

```
<tns:anything xmlns:tns='http://www.example.org/AnyComplexType'>
  <something>1</something>
</tns:anything>


template A t_product := "product";

template Anything t_attrElement_notMixed := {
  embed_values := omit,
  attr :={{String := "someattribute='1'"} {Someattr := "1"}},
  elem_list := {
    { A := t_product },
    { String := "<ns1:b xmlns:ns1='http://www.example.org/other_1'>2</ns1:b>" }
  }
}
```

Should be encoded in XML e.g. as:

```
<tns:anything xmlns:tns='http://www.example.org/AnyType' someattribute='1' someattr='1'>
  <ns:a xmlns:ns='http://www.example.org/other'>product</ns:a>
  <ns1:b xmlns:ns1='http://www.example.org/other_1'>2</ns1:b>
</tns:anything>
```

```
template Anything t_attrElement_Mixed := {
  embed_values := {"The ordered ", " has arrived ", "Wait for further information."},
  attr :={{Someattr:="1"}},
  elem_list := {
    { A := t_product },
    { String := "<ns:b xmlns:ns='http://www.example.org/other_1'>2</ns:b>" }
  }
}
```

Should be encoded in XML e.g. as:

```
<tns:anything xmlns:tns='http://www.example.org/AnyType' someattr='1'>The ordered <ns:a
  xmlns:ns='http://www.example.org/other'>product</ns:a> has arrived <ns:b
  xmlns:ns='http://www.example.org/other_1'>2</ns:b>Wait for further information.
</tns:anything>
```

7 Mapping XSD components

7.0 General

After mapping the basic layer of XML Schema (i.e. the built-in types) a mapping of the structures shall follow. Every structure that may appear, globally or not, shall have a corresponding mapping to TTCN-3.

7.1 Attributes of XSD component declarations

7.1.0 General

Tables 5 and 6 contain an overview about the use of XSD. Mappings of the attributes are described in the corresponding clauses. Tables 5 and 6 show which attributes shall be evaluated when converting to TTCN-3, depending on the XSD component to be translated.

Table 5: Attributes of XSD component declaration #1

components / attributes	element	attribute	simple type	complex type	simple content	complex content	group	wild-card
id	✓	✓	✓	✓	✓	✓	✓	
final	✓		✓	✓				
name	✓	✓	✓	✓			✓	
maxOccurs	✓ (see note 1)						✓	
minOccurs	✓ (see note 1)						✓	
ref	✓	✓					✓	
abstract	✓			✓				
block	✓			✓				
default	✓	✓						
fixed	✓	✓						
form	✓	✓						
type	✓	✓						
mixed				✓		✓		
nillable	✓							
use		✓						
substitutionGroup	✓ (see note 2)							
processContents								✓

NOTE 1: Can be used in locally defined components only.
NOTE 2: Can be used in globally defined components only.

Table 6: Attributes of XSD component declaration #2

components / attributes	all	choice	sequence	attribute Group	annotation	restriction	list	union	extension
id	✓	✓	✓	✓	✓	✓	✓	✓	✓
name				✓					
maxOccurs	✓	✓	✓						
minOccurs	✓	✓	✓						
ref				✓					

It is also necessary to consider default values for attributes coming from the original definitions of the XSD components (e.g. *minOccurs* is set to *1* for *element* components by default) when translating.

7.1.1 Id

The *id* attribute shall be ignored.

7.1.2 Ref

The *ref* attribute may reference an id or a schema component in XSD. The *ref* attribute is not translated on its own but the local *element*, *attribute*, *attributeGroup* or *group* references is mapped as specified in the appropriate clauses of the present document.

7.1.3 Name

The XSD attribute *name* holds the specified name for an XSD component. A component without this attribute shall either be defined anonymously or given by a reference (see clause 7.1.2). Names shall directly be mapped to TTCN-3 identifiers; please refer to clause 5.2.2 on constraints and properties of this conversion.

7.1.4 MinOccurs and maxOccurs

The *minOccurs* and *maxOccurs* XSD attributes provide the number of times an XSD component can appear in a context. In case of mapping locally defined XSD *element*s, *choice* and *sequence* compositors, this clause is invoked by clauses 7.3, 7.6.5 and 7.6.6.6 respectively. In case of the *all* compositor, mapping of the *minOccurs* and *maxOccurs* attributes are specified in clause 7.6.4.

The *minOccurs* and *maxOccurs* attributes of an XSD component shall be mapped together as follows:

- In the general case, when both the *minOccurs* and *maxOccurs* attribute equal to "1" (either explicitly or by defaulting to "1"), they shall be ignored, i.e. are not mapped to TTCN-3.
- If the parent of the component being translated is a *sequence* or *all*, the *minOccurs* attribute equals to "0" and the *maxOccurs* attribute equals to "1" (either explicitly or by defaulting to "1"), the TTCN-3 field resulted by mapping the respective XSD component shall be set to **`optional`**.
- In all other cases, the type of the related TTCN-3 type or field shall be set to **`record of`**, where the replicated inner type is the TTCN-3 type that would be the type of the field in the general case above. The intermediate name of the field shall be the name of XSD component being translated, postfixed with "_list", the encoding instruction "untagged" shall be attached to the outer **`record of`** and, finally, if no "untagged" encoding instruction is attached to the inner TTCN-3 type being iterated, a "name as '<*translated name*>'" encoding instruction shall be attached to the inner type, where <*translated name*> is the name resulted from applying clause 5.2.2 to the intermediate name calculated for the XSD component being translated. The **`record of`** shall be:
  - if the parent of the component being translated is a *choice*, the *minOccurs* attribute equals to "0" and the *maxOccurs* attribute equals to "1" (either explicitly or by defaulting to "1") and:
    - if the component being translated is the first direct child of the *choice* with minOccurs="0", restricted to the length range from 0 to 1;
    - if the component being translated is not the first direct child of the *choice* with minOccurs="0", restricted to the fixed length 1;
  - if the parent of the component is a *sequence* or *all*,*minOccurs* equals to "0" and *maxOccurs* equals to "unbounded", the **`record of`** shall be unrestricted;
  - if the parent of the component is a *choice*, the *minOccurs* attribute equals to "0" and the *maxOccurs* attribute is more than "1", and:
    - if the component being translated is the first direct child of the *choice* with minOccurs="0", it shall be restricted to the length range from 0 to the upper bound corresponding to the value of the *maxOccurs* attribute (where maxOccurs="unbounded" shall be translated to the upper bound infinity);
    - if the component being translated is not the first child of the *choice* with *minOccurs*="0", it shall be restricted to the length range from 1 to the upper bound corresponding to the value of the *maxOccurs* attribute (where maxOccurs="unbounded" shall be translated to the upper bound infinity);
  - if the *minOccurs* attribute does not equal to "0" and the *maxOccurs* attribute is more than "1", the **`record of`** shall be restricted to the length range corresponding to the values of the *minOccurs* and *maxOccurs* attributes (where *maxOccurs*="unbounded" shall be translated to the upper bound **`infinity`**).

NOTE 1: The effect of the "name as …" encoding instruction is, that **each repetition** of the given element in an encoded XML document will be tagged with the specified name. Thus, in this case the instruction has effect on the **elements** of the TTCN-3 **record of** field and not on the field itself.

NOTE 2: TTCN-3 constructs corresponding to anonymous XSD types always have the "untagged" encoding instruction attached before this clause is invoked.

Table 7: Summary of mapping the minOccurs and maxOccurs attributes

minOccurs	maxOccurs	in…	TTCN-3	mapping	
			TTCN-3 construct	preserved field name postfix	Is "untagged" attached?
0	0	all other cases then below			
0	1 or not present		**optional**		
1 or not present	1 or not present		<the TTCN-3 element is mandatory>		
0	unbounded		**record of** <*initial type*>	_list	yes
<*x*> ≥ 0	<*y*> ≠ 1		**record length** (<*x*>..<*y*>) **of** <*initial type*>	_list	yes
<*x*> ≥ 1 or not present	unbounded		**record length** (<*x*>..**infinity**) **of** <*initial type*> note: if minOccurs is not present <x> equals to 1	_list	yes
0	1 or not present	child of XSD choice, the first alternative with minOccurs="0"	**record length (**0..1**) of** <initial type>	_list	yes
0	unbounded		**record of** <initial type>	_list	yes
0	1 or not present	child of XSD choice, not the first alternative with minOccurs="0"	**record length (**1**) of** <initial type>	_list	yes
0	unbounded		**record length (**1..infinity**)of** <initial type>	_list	yes

EXAMPLE 1: Mapping of an optional *element*:

```
<xsd:complexType name="e15a">
    <xsd:sequence>
        <xsd:element name="foo" type="xsd:integer" minOccurs="0"/>
        <xsd:element name="bar" type="xsd:float"/>
    </xsd:sequence>
</xsd:complexType>
```

Is translated to an optional field e.g. as:

```
type record E15a {
    XSD.Integer foo optional,
    XSD.Float  bar
}
with {
    variant "name as uncapitalized";
}
```

EXAMPLE 2: Mapping of *element*s allowing multiple recurrences:

The unrestricted case:

```
<xsd:complexType name="e15b">
    <xsd:sequence>
        <xsd:element name="foo" type="xsd:integer" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="bar" type="xsd:float"/>
    </xsd:sequence>
</xsd:complexType>
```

Is translated to TTCN-3 e.g. as:

```
type record E15b {
    record of XSD.Integer foo_list,
    XSD.Float  bar
}
with {
    variant "name as uncapitalized";
    variant(foo_list) "untagged";
    variant(foo_list[-]) "name as 'foo'";
}
```

The length restricted case:

```
<xsd:complexType name="e15c">
    <xsd:sequence>
        <xsd:element name="foo" type="xsd:integer" minOccurs="5" maxOccurs="10"/>
        <xsd:element name="bar" type="xsd:float"/>
    </xsd:sequence>
</xsd:complexType>
```

Is translated to TTCN-3 e.g. as:

```
type record E15c {
    record length(5..10) of XSD.Integer foo_list,
    XSD.Float bar
}
with {
    variant "name as uncapitalized ";
    variant(foo_list) "untagged";
    variant(foo_list[-]) "name as 'foo'";
}
```

EXAMPLE 3: Mapping of *a group* reference:

Provided:

```
<xsd:group name="foobarGroup">
  <xsd:sequence>
    <xsd:element name="foo" type="xsd:string"/>
    <xsd:element name="bar" type="xsd:string"/>
  </xsd:sequence>
</xsd:group>
```

The optional case:

```
<xsd:complexType name="e15d">
  <xsd:group ref="ns:foobarGroup" minOccurs="0"/>
</xsd:complexType>
```

Is translated to TTCN-3 e.g. as:

```
type record FoobarGroup {
    XSD.String foo,
    XSD.String bar
}
with {
    variant "untagged";
}
```

(Please note, no "name as..." instruction is attached to the type due to the presence of the untagged instruction.)

```
type record E15d {
    FoobarGroup foobarGroup optional
}
with {
    variant "name as uncapitalized";
}
```

EXAMPLE 4: Mixed case, both *element*s and *a group* reference are present:

```
<xsd:complexType name="e15f">
    <xsd:sequence>
        <xsd:element name="comment" minOccurs="0" maxOccurs="unbounded" type="xsd:string"/>
        <xsd:group ref="ns:foobarGroup" minOccurs="5" maxOccurs="10"/>
    </xsd:sequence>
</xsd:complexType>
```

Is translated to TTCN-3 e.g. as:

```
type record E15f {
    record of XSD.String comment_list,
    record length (5..10) of FoobarGroup foobarGroup_list
}
with {
    variant "name as uncapitalized ";
    variant(comment_list) "untagged";
```

```
    variant(comment_list[-]) "name as 'comment'";
    variant(foobarGroup_list) "untagged";
    //pls. note, no "name as..." instruction is attached to foobarGroup[-] due to the
    //presence of the "untagged" instruction attached to the FoobarGroup type.
}
```

EXAMPLE 5: Resolving a name clash:

The Schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema  xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:ns="www.example.org/name_clash_element-attribute"
             targetNamespace="www.example.org/name_clash_element-attribute">

    <xsd:simpleType name="start_list">
        <xsd:list itemType="xsd:string"/>
    </xsd:simpleType>

    <xsd:complexType name="start">
        <xsd:sequence>
            <xsd:element name="start" type="xsd:integer" minOccurs="0" maxOccurs="10"/>
        </xsd:sequence>
        <xsd:attribute name="start_list" type="ns:start_list"/>
    </xsd:complexType>
</xsd:schema>
```

Is translated to the TTCN-3 module e.g. as:

```
module http_www_example_org_name_clash_element_attribute {

    import from XSD all;

    type record of XSD.String Start_list
    with {
        variant "name as uncapitalized";
        variant "list";
    }

    type record Start {
        Start_list start_list optional,
        record length(0 .. 10) of XSD.Integer start_list_1
        //the composed name of the record of field would clashes with the name of the field
        //added for the XSD attribute, this is resolved by postfixing it according to $5.2.2
    }
    with {
        variant "name as uncapitalized";
        variant (start_list) "attribute";
        variant (start_list_1) "untagged";
        variant (start_list_1[-]) "name as 'start'";
    };

}
with {
    encode "XML";
    variant "namespace as 'http://www.example.org/name_clash_element-attribute' prefix 'ns'";
    variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

EXAMPLE 6: Mapping of childs of choice components:

The XSD elements:

```
<xsd:element name="ChoiceChildMinMax">
  <xsd:complexType>
    <xsd:choice>
        <xsd:element name="elem0" type="xsd:string" minOccurs="1" maxOccurs="5"/>
        <xsd:element name="elem1" type="xsd:string" minOccurs="0" />
        <xsd:element name="elem2" type="xsd:string" minOccurs="0" />
        <xsd:element name="elem3" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>

<xsd:element name="minOccurs_maxOccurs_frame">
    <xsd:complexType>
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
```

```
            <xsd:element ref="ns:ChoiceChildMinMax"/>
        </xsd:choice>
    </xsd:complexType>
</xsd:element>
```

Are translated to TTCN-3 e.g. as:

```
type record ChoiceChildMinMax {
    union {
        record length(1 .. 5) of XSD.String elem0_list,
            // child of choice with minOccurs different from 0
        record length(0 .. 1) of XSD.String elem1_list,
            // first child of choice with minOccurs 0;
            // this alternative is to be used create an empty choice element
        record length(1) of XSD.String elem2_list,
            // second child of choice with minOccurs 0
        record length(1 .. infinity) of XSD.String elem3_list
            // third child of choice with minOccurs 0
    } choice
}
with {
    variant "element";
    variant (choice) "untagged";
    variant (choice.elem0_list) "untagged";
    variant (choice.elem0_list[-]) "name as 'elem0'";
    variant (choice.elem1_list) "untagged";
    variant (choice.elem1_list[-]) "name as 'elem1'";
    variant (choice.elem2_list) "untagged";
    variant (choice.elem2_list[-]) "name as 'elem2'";
    variant (choice.elem3_list) "untagged";
    variant (choice.elem3_list[-]) "name as 'elem3'";
};

type record MinOccurs_maxOccurs_frame {
    record of union {
        ChoiceChildMinMax choiceChildMinMax
    } choice_list
}
with {
    variant "name as uncapitalized";
    variant "element";
    variant (choice_list) "untagged";
    variant (choice_list[-]) "untagged";
    variant (choice_list[-].choiceChildMinMax) "name as capitalized";
};
```

And the TTCN-3 template:

```
template MinOccurs_maxOccurs_frame t_MinOccurs_maxOccurs_inChoice := {
    choice_list := {
        // instances of the element elem0
    { choiceChildMinMax := { choice := { elem0_list := {"e01", "e02" }}}},
        // an instance of the element elem1
    { choiceChildMinMax := { choice := { elem1_list := { "e1" }}}},
        // an instance of the element elem2
    { choiceChildMinMax := { choice := { elem2_list := { "e2" }}}},
        // instances of the element elem3
    { choiceChildMinMax := { choice := { elem3_list := { "e31", "e32", "e33" }}}},
        // an empty choice element
    { choiceChildMinMax := { choice := { elem1_list := {}}}}
  }
}
```

Could be encoded in XML e.g. as:

```
<?xml version="1.0" encoding="UTF-8"?>
<this:minOccurs_maxOccurs_frame xmlns:this="http://www.example.org/minOccurs_maxOccurs"
                          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                          xsi:schemaLocation="http://www.example.org/minOccurs_maxOccurs
                                              ../XSD/minOccurs_maxOccurs.xsd">
<!-- instances of the element elem0 -->
<this:ChoiceChildMinMax><elem0>e01</elem0><elem0>e02</elem0></this:ChoiceChildMinMax>

<!-- an instance of the element elem1 -->
<this:ChoiceChildMinMax><elem1>e1</elem1></this:ChoiceChildMinMax>

<!-- an instance of the element elem2 -->
<this:ChoiceChildMinMax><elem2>e2</elem2></this:ChoiceChildMinMax>
```

```
<!-- instances of the element elem3 -->
<this:ChoiceChildMinMax><elem3>e31</elem3><elem3>e32</elem3><elem3>e33</elem3>
</this:ChoiceChildMinMax>

<!— an empty choice element -->
<this:ChoiceChildMinMax/>

</this:minOccurs_maxOccurs_frame>
```

7.1.5 Default and Fixed

The XSD *default* attribute assigns a default value to a component in cases where it is missing in the XML data.

The XSD *fixed* attribute gives a fixed constant value to a component according to the given type, so in some XML data the value of the component may be omitted. The XSD *fixed* attribute can also be applied to XSD facets, preventing a derivation of that type from modifying the value of the fixed facets.

As *default* has no equivalent in TTCN-3 space, it shall be mapped to a "defaultForEmpty …" encoding instruction. The *fixed* attribute applied to *attribute* or *element* elements shall be mapped to a subtype definition with the single allowed value identical to the value of the *fixed* attribute plus a "defaultForEmpty …" encoding instruction identifying the value of the fixed attribute as well. The *fixed* attribute applied to XSD facets shall be ignored.

EXAMPLE:

```
<xsd:element name="elementDefault" type="xsd:string" default="defaultValue"/>
<xsd:element name="elementFixed" type="xsd:string" fixed="fixedValue"/>
```

Will be translated to TTCN-3 e.g. as:

```
type XSD.String ElementDefault
with {
    variant "element";
    variant "defaultForEmpty as 'defaultValue'";
    variant "name as uncapitalized";
}

type XSD.String ElementFixed ("fixedValue")
with {
    variant "element";
    variant "defaultForEmpty as 'fixedValue'";
    variant "name as uncapitalized"
}
```

7.1.6 Form

The XSD *form* attribute controls if an attribute or element tag shall be encoded in XML by using a qualified or unqualified name. The values of the *form* attributes shall be preserved in the "form as…" encoding instructions as specified below:

a) If the value of the *form* attribute is *qualified* and the *attributeFormQualified* encoding instruction is attached to the TTCN-3 module the given XSD declaration contributes to, or the value of the *form* attribute is *unqualified* and no *attributeFormQualified* encoding instruction is assigned to the corresponding TTCN-3 module, the *form* attribute shall be ignored.

b) If the value of a *form* attribute of an XSD *attribute* declaration is *qualified* and no *attributeFormQualified* encoding instruction is attached to the target TTCN-3 module, or the value of a *form* attribute of an *element* declaration is *qualified* and no *elementFormQualified* encoding instruction is attached to the target TTCN-3 module, a **`"form as qualified"`** encoding instruction shall be attached to the TTCN-3 field resulted from mapping the given XSD *attribute* or *element* declaration.

c) If the value of a *form* attribute of an XSD *attribute* declaration is *unqualified* and the *attributeFormQualified* encoding instruction is attached to the target TTCN-3 module, or the value of a *form* attribute of an *element* declaration is *unqualified* and the *elementFormQualified* encoding instruction is attached to the target TTCN-3 module, a **`"form as unqualified"`** encoding instruction shall be attached to the TTCN-3 field resulted from mapping the given XSD *attribute* or *element* declaration.

NOTE: An XSD declaration may contribute to more than one TTCN-3 module (see clause 5.1), therefore in case of a given XSD declaration item a) and b) or c) above may apply at the same time.

Table 8 summarizes the mapping of the *attributeFormDefault*, *elementFormDefault* (see also clause 5.1) and *form* XSD attributes.

Table 8: Summary of mapping of the *form* XSD attribute

				"namespace as" encoding instruction attached to the target	**attributeFormQualified and/or elementFormQualified encoding instructions attached to the target TTCN-3 module**	
				TTCN-3 module	**absent**	**present**
attributeFormDefault and/or elementFormDefault in the ancestor schema element	**any value or absent**	***form* attribute**	**any value or absent**	**absent**	"form as…" absent	N/A (see note)
	unqualified or absent	***form* attribute**	**absent**	**present**	"form as…" absent	"form as unqualified"
			unqualified	**present**	"form as…" absent	"form as unqualified"
			qualified	**present**	"form as qualified"	"form as…" absent
	qualified	***form* attribute**	**absent**	**present**	N/A (see note)	"form as…" absent
			unqualified	**present**	N/A (see note)	"form as unqualified"
			qualified	**present**	N/A (see note)	"form as…" absent
NOTE: Excluded by the mapping of attributeFormDefault and elementFormDefault in clause 5.1.						

7.1.7 Type

The XSD *type* attribute holds the type information of the XSD component. The value is a reference to the global definition of *simpleType*, *complexType* or built-in type. If *type* is not given, the component shall define either an anonymous (inner) type, or contain a reference attribute (see clause 7.1.2), or use the XSD ur-type definition.

7.1.8 Mixed

The *mixed* content attribute allows inserting text between the elements of XSD complex type or element definitions. Its translation is defined in clause 7.6.8.

7.1.9 Abstract

The *abstract* XSD attribute can be used in global *element* XSD element information items and *complexType* XSD element information items. When its value is set to *"true"* in a global *element* XSD definition, the given element shall not be used in instances of the given XML Schema but is forced to be substituted with a member *element* of the substitution group of which the abstract element is the head of (if there is no substitutable elements in the Schema, the element cannot be used in instance documents). When its value is set to *"true"* in a global *complexType* XSD definition, XSD elements referencing this type in their *type* attribute are forced to be instantiated by using an another type definition, which is derived from the abstract type (the actual type used at instantiation shall be indicated by the xsi:type XML attribute in the instance of the given element). See more details on mapping of substitutions in clause 8.

The *abstract* XSD attribute shall be translated to TTCN-3 by adding the "abstract" encoding instruction to the generated TTCN-3 type definition corresponding to the XSD *element* or *complexType* information items with the *abstract* attribute value *"true"*. If the value of the *abstract* attribute information item is set to *"false"* directly or indirectly (i.e. by defaulting to *"false"*), the *abstract* XSD attribute shall be ignored. See example in clause 8.1.1.

7.1.10 Block and blockDefault

The XSD *block* and *blockDefault* attribute information items control the allowed element and type substitutions at the instance level; *blockDefault* can be used in XSD *schema* elements, and has effect on all element and type child of the schema. This default value can be overridden by a *block* attribute applied to a given *element* or *complexType* element information item directly. This will result produce the effective block value for the given *element* or *complexType*. See also clauses 3.3.2 and 3.4.2 of XML Schema Part 1 [9].

The effective block value shall be translated together with substitution. If a TTCN-3 code allowing element substitutions is generated (see clause 8), the effective block value of head elements shall be translated together with the head element of the substitution group according to clause 8.1.1. If a TTCN-3 code allowing type substitutions is generated (see clause 8), the effective block value of substitutable parent types shall be translated together with the substitutable parent types according to clause 8.2. The *blockDefault* and *block* attributes shall be ignored in all other cases.

7.1.11 Nillable

If the *nillable* attribute of an *element* declaration is set to *"true"*, then an element may also be valid if it carries the namespace qualified attribute with (local) name *nil* from the namespace "http://www.w3.org/2001/XMLSchema-instance" and the value *"true"* (instead of a value of its type).

A simple-type nillable XSD *element* shall be mapped to a TTCN-3 **`record`** type (in case of global elements) or field (in case of local elements), with the name resulted by applying clause 5.2.2 to the name of the corresponding element. The **`record`** type or field shall contain one **`optional`** field with the name "content" and its type shall be the TTCN-3 type equivalent of the element's type. The **`record`** type or field added shall be appended with the "useNil" encoding instruction.

EXAMPLE 1: Mapping of simple-type *nillable* elements:

```
<xsd:element name="remarkNillable" type="xsd:string" nillable="true"/>

<xsd:element name="e16c">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="foo" type="xsd:integer"/>
            <xsd:element name="bar" type="xsd:string" nillable="true"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Will be translated to TTCN-3 e.g. as:

```
type record RemarkNillable {
    XSD.String content optional
}
with {
    variant "name as uncapitalized";
    variant "element";
    variant "useNil"
}

type record E16c {
    XSD.Integer foo,
    record {
        XSD.String content optional
    } bar
}
with {
    variant "name as uncapitalized";
    variant "element";
    variant(bar) "useNil"
}
```

Which allows e.g. the following encoding:

```
template E16c t_E16c :=
{
    foo:=3,
    bar:= { content := omit }
```

```
}

<?xml version="1.0" encoding="UTF-8"?>
    <e16c>
        <foo>3</foo>
        <bar xsi:nil="true"/>
    </e16c>
```

ComplexType XSD *elements* are mapped to an outer **record** type or field, according to clause 7.6. When the *nillable* XSD attribute with the value *'true'* is contained in such an *element*, the *element* shall be mapped to an extra optional TTCN-3 **record** field with the name resulted from applying clause 5.2.2 to "content". The attributes of the *complexType* element shall be mapped fields of the outer record, i.e. they shall not be the members of the inner record, generated due to the *nillable* XSD attribute. The outer **record**, corresponding to the *complexType* component of the nillable element, shall be appended with the "useNil" encoding instruction.

EXAMPLE 2: Mapping of complex-type *nillable* elements and joint use of attributes:

```
<xsd:element name="remark" type="xsd:string" nillable="true"/>

<xsd:element name="SeqNillable" nillable="true">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="forename" type="xsd:string" nillable="true"/>
            <xsd:element name="surname" type="xsd:string" minOccurs="0" nillable="true"/>
            <xsd:element name="livingAddress"   type="xsd:string" minOccurs="0"
                         maxOccurs="unbounded"  nillable="true"/>
            <xsd:element ref="ns:remark"/>
        </xsd:sequence>
        <xsd:attribute name="foo" type="xsd:integer"/>
        <xsd:attribute name="bar" type="xsd:integer"/>
    </xsd:complexType>
</xsd:element>
```

Is translated to TTCN-3 e.g. as:

```
type record Remark {
    XSD.String content optional
}
with {
    variant "name as uncapitalized";
    variant "element";
    variant "useNil"
}

type record SeqNillable {
        XSD.Integer bar optional,
        XSD.Integer foo optional,
    //record generated for the nillable attribute of the top element "SeqNillable"
    record {
        //mapping of the contained simple-type nillable element "forename"
        record {
            XSD.String content optional
        } forename,
        //mapping of the contained simple-type nillable element "surname"
        record {
            XSD.String content optional
        } surname optional,
        //mapping of the contained simple-type nillable element "livingAddress"
        record of record {
            XSD.String content optional
        } livingAddress_list,
        //mapping of the referenced nillable element "remark":
        //nillable attribute is resolved in the referenced type
        Remark remark
    } content optional
}
with {
    variant "element";
    variant "useNil";
    variant (bar, foo) "attribute"; variant(content.livingAddress_list) "untagged";
    variant (content.livingAddress_list[-]) "name as'livingAddress'";
    variant (content.forename, content.surname, content.livingAddress_list[-])
            "useNil";
}
```

7.1.12 Use

XSD local attribute declarations and references may contain also the special attribute *use*. The *use* attribute specifies the presence of the attribute in an XML value. The values of this attribute are: *optional*, *prohibited* and *required* with the default value *optional*. If the *use* attribute is missing or its value is *optional* in an XSD attribute declaration, the TTCN-3 field resulted by the mapping of the corresponding attribute shall be **`optional`**. If the value of the *use* attribute is *required*, the TTCN-3 field corresponding to the XSD attribute shall be mandatory (i.e. without **`optional`**). XSD attributes with the value of the *use* attribute *prohibited* shall not be translated to TTCN-3.

EXAMPLE: Mapping of the *use* attribute:

```
<xsd:complexType name="e17a">
    <xsd:sequence>
    </xsd:sequence>
    <xsd:attribute name="fooLocal" type="xsd:float" use="required" />
    <xsd:attribute name="barLocal1" type="xsd:string" />
    <xsd:attribute name="barLocal2" type="xsd:string" use="optional" />
    <xsd:attribute name="dingLocal" type="xsd:integer" use="prohibited" />
</xsd:complexType>
```

Is translated to TTCN-3 e.g. as:

```
type record E17a {
    XSD.String barLocal1        optional,
    XSD.String barLocal2        optional,
    XSD.Float fooLocal
}
with {
    variant "name as uncapitalized ";
    variant (barLocal1, barLocal2, fooLocal) "attribute";
}
```

7.1.13 Substitution group

The XSD *substitutionGroup* attribute can be used in global XSD *element* information items. Its value is the name of the head element of a substitutionGroup and thus the XSD *element* definition containing the *substitutionGroup* attribute becomes a member of that substitution group.

The *substitutionGroup* attribute information item shall be ignored when the *element* is translated to TTCN-3.

NOTE: See more details on mapping XSD substitutions in clause 8.

7.1.14 Final

The *final* XSD attribute information item constrains the creation of derived types and types of substitution group members (see more details on mapping of substitutions in clause 8).

The *final* XSD attribute information item(s) shall produce no TTCN-3 language construct when translating an XML Schema to TTCN-3.

NOTE: As specified in clause 5, the XML Schema is validated before the actual translation process can be started. Therefore the restrictions imposed by any *final* attribute(s) will be enforced during schema validation and no need to reflect it in the generated TTCN-3 code.

7.1.15 Process contents

The *processContents* XSD attribute information item controls the validation level of the content of instances corresponding to XSD *any* and *anyAttribute* information items (see clause 7.7). Its allowed values are *"strict"*, *"lax"* and *"skip"*. This attribute shall be translated by attaching a "processContents …" encoding instruction replicating the value of the XSD attribute to the TTCN-3 component generated for the XSD element with the *processContents* XSD attribute.

If the value of the *processContents* XSD attribute is "strict", and no XSD *schema* is present with a target namespace allowed by the *namespace* attribute of the XSD *any* or *anyAttribute* element being translated, or the *schema* does not contain an XSD *element* or *attribute* declaration respectively, this shall cause an error.

7.2 Schema component

The *schema* element information items are not directly translated to TTCN-3 but the content(s) of schema element information item(s) with the same target namespace (including absence of the target namespace) are mapped to definitions of a target TTCN-3 module. See more details in clause 5.1.

7.3 Element component

An XSD *element* component defines a new XML element. Elements may be global (as a child of either *schema* or *redefine*), in which case they are obliged to contain a name attribute or may be defined locally (as a child of *all*, *choice* or *sequence)* using a *name* or *ref* attribute.

Globally defined XSD *element*s shall be mapped to TTCN-3 type definitions. In the general case, when the *nillable* attribute of the element is "false" (either explicitly or by defaulting to "false"), the type of the TTCN-3 type definition shall be one of the following:

a) In case of XSD datatypes, and simple types defined locally as child of the *element*, the type of the XSD *element* mapped to TTCN-3.

b) In case of XSD user-defined types referenced by the *type* attribute of the *element*, the TTCN-3 type generated for the referenced XSD type.

c) In case the child of the *element* is a locally defined *complexType*, it shall be a TTCN-3 **`record`**.

d) If none of the above cases apply and the element has the *substitutionGroup* attribute, it shall be the type of the head element of the substitution group.

e) Otherwise it shall be the type XSD.AnyType or XSD.AnyComplexType (see clauses 6.8 and B.3.1).

NOTE: In the last case the element's type defaults to the ur-type definition in XSD, see clause 3.3.2 of [5].

The name of the TTCN-3 type definition shall be the result of applying clause 5.2.2 to the *name* of the XSD *element*. When *nillable* attribute is "true", the procedures in clause 7.1.11 shall be invoked. The encoding instruction "element" shall be appended to the TTCN-3 type definition resulted by mapping of a global XSD *element*.

EXAMPLE 1: Mapping of a globally defined element:

```
<xsd:element name="e16a" type="typename"/>
```

Is translated to TTCN-3 e.g. as:

```
type Typename E16a
with {
    variant "element";
    variant "name as uncapitalized";
}
```

Locally defined *element*s shall be mapped to fields of the enframing type or structured type field. In the general case, when both the *minOccurs* and *maxOccurs* attribute equal to "1" (either explicitly or by defaulting to "1") and the *nillable* attribute of the element is "false" (either explicitly or by defaulting to "false"), the type of the field shall be:

- the type reference of the TTCN-3 type definition, resulted by mapping the XSD *type* that is the value of the *type* attribute of the element, or
- the type resulted by mapping the locally defined type of the XSD *element*, mapped according to the rules specified for global *element*s in this clause above, and
- the name of the field shall be the result of applying clause 5.2.2 to the name of the XSD *element*.

When a local element is defined by reference (the *ref* attribute is used) and the target namespace of the XSD Schema in which the referenced *element* is defined differs from the target namespace of the referencing XSD Schema (including the no target namespace case), the TTCN-3 field generated for this *element* reference shall be appended with a "namespace as" encoding instruction (see clause B.3.1), which shall identify the namespace and optionally the prefix of the XSD schema in which the referenced entity is defined.

When either the *minOccurs* or the *maxOccurs* attributes or both differ from "1", the procedures in clause 7.1.4 shall be invoked.

When the *nillable* attribute is "true", the procedures in clause 7.1.11 shall be invoked.

EXAMPLE 2: Mapping of locally defined elements, general case (see further examples in clauses 7.1.4 and 7.1.11):

```
<xsd:complexType name="e16b">
    <xsd:sequence>
        <xsd:element name="foo" type="xsd:integer"/>
        <xsd:element name="bar" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
```

Is translated to TTCN-3 e.g. as:

```
type record E16b
{
    XSD.Integer foo,
    XSD.String bar
}
with {
    variant "name as uncapitalized";
}
```

7.4 Attribute and attribute group definitions

7.4.1 Attribute element definitions

Attribute elements define valid qualifiers for XML data and are used when defining complex types. Just like XSD *element*s, *attribute*s can be defined globally (as a child of *schema* or *redefine*) and then be referenced from other definitions or defined locally (as a child of *complexType*, *restriction*, *extension* or *attributeGroup*) without the possibility of being used outside of their context.

Global attributes shall be mapped to TTCN-3 type definitions. In the general case, the type of the TTCN-3 type definition shall be one of the following:

a) In case of XSD datatypes, and simple types defined locally as child of the *attribute* element, the type of the XSD *attribute* mapped to TTCN-3.

b) In case that a XSD user-defined type is referenced by the *type* attribute of the XSD *attribute* element, the TTCN-3 type generated for the referenced XSD type.

c) Otherwise it shall be the type XSD.AnySimpleType (see clauses 6.8 and B.3.1).

NOTE: In the last case the element's type defaults to the simple ur-type definition in XSD, see clause 3.2.2 of [5].

The name of the TTCN-3 type definition shall be the result of applying clause 5.2.2 to the *name* of the XSD *attribute* element. The generated TTCN-3 type definition shall be appended with the "attribute" TTCN-3 encoding instruction.

For the mapping of locally defined attributes please refer to clause 7.6.7.

EXAMPLE: Mapping of a globally defined attribute:

```
<xsd:attribute name="e17" type="typename"/>
```

Is mapped to TTCN-3 e.g. as:

```
type Typename E17
with {
    variant "attribute";
    variant "name as uncapitalized";
}
```

7.4.2 Attribute group definitions

An XSD *attributeGroup* defines a group of attributes that can be included together into other definitions by referencing the *attributeGroup*. As children *attribute* elements of *attributeGroup* definitions are directly mapped to the TTCN-3 record types corresponding to the *complexType* referencing the *attributeGroup*, *attributeGroup*-s are not mapped to TTCN-3. See also clauses 7.6.1 and 7.6.7.

7.5 SimpleType components

7.5.0 General

XSD simple types may be defined globally (as child of `schema` and using a mandatory `name` attribute) or locally (as a child of `element`, `attribute`, `restriction`, `list` or `union`) in a named or anonymous fashion. The `simpleType` components are used to define new simple types by three means:

- Restricting a built-in type (with the exception of anyType, anySimpleType) by applying a facet to it.
- Building lists.
- Building unions of other simple types.

These means are quite different in their translation to TTCN-3 and are explained in the following clauses. For the translation of attributes for simple types please refer to the general mappings defined in clause 7.1. Please note that an XSD `simpleType` is not allowed to contain elements or attributes, redefinition of these is done by using XSD `complexType`-s (see clause 7.6).

7.5.1 Derivation by restriction

For information about restricting built-in types, please refer to clause 6 which contains an extensive description on the translation of restricted *simpleType* using facets to TTCN-3.

If the definition of a new named or unnamed simple type uses another simple type as the base of the restriction without changing the base type (i.e. no facet is applied), it shall be translated to a TTCN-3 type synonym, completed with necessary additional encoding instructions, to the base type (see clause 6.4 of ETSI ES 201 873-1 [1]).

NOTE: This means that tools need not analyse the effective value space of the base and the derived types, but can make a decision based on the presence of facet(s) in the derived type.

It is also possible in XSD to restrict an anonymous simple type. The translation follows the mapping for built-in data types, but instead of using the base attribute to identify the type to apply the facet to, the base attribute type shall be omitted and the type of the inner, anonymous simpleType shall be used.

EXAMPLE 1: Definition of a *simpleType* without changing the base type:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.example.org/aliases"
    xmlns:ns=http://www.example.org/aliases>

    <xsd:simpleType name="simple-base">
        <xsd:restriction base="xsd:integer"/>
    </xsd:simpleType>

    <xsd:simpleType name="simple-restr">
        <xsd:restriction base="ns:simple-base"/>
    </xsd:simpleType>

    <xsd:element name="elem-simple-restr">
        <xsd:simpleType>
            <xsd:restriction base="ns:simple-base"/>
        </xsd:simpleType>
        </xsd:element>
</xsd:schema>
```

Will translate to the TTCN-3 module, e.g. as:

```
module http_www_example_org_aliases {

    import from XSD all;

    type XSD.Integer Simple_base
    with {
        variant "name as 'simple-base'";
    };

    type Simple_base Simple_restr
    with {
        variant "name as 'simple-restr'";
    };

    type Simple_base Elem_simple_restr
    with {
      variant "name as 'elem-simple-restr'";
      variant "element";
    };
}
with {
  encode "XML";
  variant "namespace as 'http://www.example.org/aliases' prefix 'ns'";
  variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

EXAMPLE 2: Restricting an anonymous *simpleType* using a pattern facet:

```
<?xml version="1.1" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:ns="http://www.example.org/simpleType"
targetNamespace="http://www.example.org/simpleType">
    <xsd:element name="e18">
        <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                <xsd:pattern value="(aUser|anotherUser)@(i|I)nstitute"/>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:element>
</xsd:schema>
```

Will result e.g. in the following TTCN-3 module:

```
module http_www_example_org_simpleType {
    import from XSD all;
    type XSD.String E18 (pattern "(aUser|anotherUser)@(i|I)nstitute")
    with {
        variant "name as uncapitalized";
        variant "element";
    }
}
with {
    encode "XML";
    variant "namespace as 'http://www.example.org/simpleType' prefix 'ns'";
    variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

7.5.2 Derivation by list

XSD list components shall be mapped to the TTCN-3 *record of* type. In their simplest form, when the i*temType* attribute identifies the base type of the derivation, the replicated type of the TTCN-3 record of shall be the type mapped from the XSD type referenced by *itemType*.

When the XSD *list* is used to derive a list type from an (embedded) unnamed XSD type, first the type included by the *list* start and end tags shall implicitly be translated to TTCN-3 and this type shall be the replicated type of the generated TTCN-3 **`record of`**.

Finally, the encoding instruction "list" shall be applied to the generated **`record of`** type.

When using any of the supported XSD facets (length, maxLength, minLength) the translation shall follow the mapping for built-in list types, with the difference that the base type shall be determined by an anonymous inner list item type.

The other XSD facets shall be mapped accordingly (refer to respective 6.1 clauses). If no *itemType* is given, the mapping has to be implemented using the given inner type (see clause 7.5.3).

EXAMPLE 1: Mapping a list derived by using the i*temType* attribute:

```
<xsd:simpleType name="e19">
    <list itemType="float"/>
</xsd:simpleType>
```

Will be translated to TTCN-3 e.g. as:

```
type record of XSD.Float E19
with {
    variant "list";
    variant "name as uncapitalized"
}
```

EXAMPLE 2: Mapping an unnamed simple union type derived by list:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:tns="http://www.example.org/list_union"
            targetNamespace="http://www.example.org/list_union" >

<xsd:element name="MyUnionList">
    <xsd:simpleType>
      <xsd:list>
        <xsd:simpleType>
          <xsd:union>
            <xsd:simpleType>
              <xsd:restriction base="xsd:boolean" />
            </xsd:simpleType>
            <xsd:simpleType>
              <xsd:restriction base="xsd:float" />
            </xsd:simpleType>
          </xsd:union>
        </xsd:simpleType>
      </xsd:list>
    </xsd:simpleType>
  </xsd:element>

</xsd:schema>
```

Is translated to TTCN-3 e.g. as (translation of the embedded union type see in the next clause):

```
module http_www_example_org_list_union {
  import from XSD all;

  type record of union
  {
    XSD.Boolean alt_,
    XSD.Float alt_1
  } MyUnionList
  with {
    variant "list";
    variant "element";
    variant ([-]) "useUnion";
    variant ([-].alt_) "name as ''";
    variant ([-].alt_1) "name as ''";
  };
}
with {
  encode "XML";
  variant "namespace as 'http://www.example.org/list_union' prefix 'tns'";
  variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

EXAMPLE 3: Consider this example:

```
<xsd:element name="e20">
    <xsd:simpleType>
        <xsd:restriction>
            <xsd:simpleType>
                <xsd:list itemType="float"/>
            </xsd:simpleType>
            <xsd:length value="3"/>
```

```
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

Will map to TTCN-3 e.g. as:

```
type record length(3) of XSD.Float E20
with {
    variant "name as uncapitalized";
    variant "element";
    variant "list";
}
```

For instance the template:

```
template E20 t_E20:={ 1.0, 2.0, 3.0 }
```

Can be encoded in XML, for example, as:

```
<?xml version="1.0" encoding="UTF-8"?><e20>1.0 2.0 3.0</e20>
```

7.5.3 Derivation by union

An XSD union is considered as a set of mutually exclusive alternative types for a *simpleType*. As this is compatible with the *union* type of TTCN-3, a *simpleType* derived by *union* in XSD shall be mapped to a union type definition in TTCN-3. The generated TTCN-3 **union** type shall contain one alternative for each member type of the XSD *union*, preserving the textual order of the member types in the initial XSD union type. The field names of the TTCN-3 **union** type shall be the result of applying clause 5.2.2 to either to the unqualified name of the member type (in case of built-in XSD data types and user defined named types) or to the string "alt" (in case of unnamed member types).

In case of a restriction of an existing base type using enumerated values the resulting TTCN-3 type for each value will be a *record* with two fields. The first field is an enumeration of all types from the union members and the field name "xsiType", the second field is an enumeration of all enumerated values and the field name "content". The derivation of the enumeration value identifiers follows the rules from clause 5. The names of the xsiType enumerated values should be the same names as the alternatives in the other union-mappings (i.e. if it is a union containing anonymous simple types, it should be alt_, alt_1 etc. for these, only for referenced types (in the memberTypes attribute), the referenced type names shall be used - same as in other union).

NOTE 1: The procedure allows multiple combinations for enumerated values from the XSD restriction. E.g. the enumerated value "x20" may indicate an **integer** or **float** value.

XSD requires (see XML Schema Part 2: Datatypes [9], clause 2.5.1.3) that an element or attribute value of an instance is validated against the member types in the order in which they appear in the XSD definition until a match is found (considering any xsi:type attribute present, see also clause B.3.24). A TTCN-3 tool has to use this strategy as well, when decoding an XSD *union* value.

The encoding instruction "useUnion" shall be applied to the generated **union** type and, in addition, the "name as ''" ("name as followed by a pair of single quote followed by a double quote) encoding instruction shall be applied to each field generated for an unnamed member type.

When restricted by an enumeration facet the encoding instruction "useUnion" shall be applied to the generated **record** type and, in addition, "text 'name' as 'freetext'" encoding instruction shall be applied to the "xsiType" and "content" fields for the name(s) of one or more TTCN-3 enumeration values is(are) differs from the related XSD enumeration item.

NOTE 2: Please note, that alt and the names of several built-in XSD data types are TTCN-3 keywords, hence according to the naming rules these field identifiers will be postfixed with a single underscore character.

The only supported facet is *enumeration*, allowing mixing enumerations of different kinds.

EXAMPLE 1: Mapping of named simple type definitions derived by union:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:ns=http://www.example.org/union
targetNamespace="http://www.example.org/union">
```

```
    <xsd:simpleType name="e21memberlist">
        <xsd:union memberTypes="xsd:integer xsd:boolean xsd:string "/>
    </xsd:simpleType>

    <xsd:element name="e21namedElement" type="ns:e21memberlist"/>

    </xsd:schema>
```

Results e.g. in the following mapping:

```
module http_www_example_org_union {

     import from XSD all;

     type E21memberlist E21namedElement
     with {
     variant "name as uncapitalized";
     variant "element";
 }

     type union E21memberlist {
         XSD.Integer integer_,
         XSD.Boolean boolean_,
         XSD.String string
     }
     with {
         variant "name as uncapitalized";
         variant "useUnion";
         variant (integer_) "name as 'integer'";
         variant (boolean_) "name as 'boolean'";
     }
 }
 with {
     encode "XML";
     variant "namespace as 'http://www.example.org/union' prefix 'ns'";
     variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
 }
```

For instance, the below structure:

```
 template E21namedElement t_UnionNamedInt := { integer_ := 1 }
```

Should result e.g. in the following XML encoding:

```
    <?xml version="1.0" encoding="UTF-8"?>
    <ns:e21namedElement xmlns:ns='http://www.example.org/union'
xmlns:xsd='http://www.w3.org/2001/XMLSchema'    xmlns:xsi='http://www.w3.org/2001/XMLSchema-
instance' xsi:type= 'xsd:integer'>1</ns:e21namedElement>
```

EXAMPLE 2: Mapping of unnamed simple type definitions derived by union (compare it with the previous example):

```
    <?xml version="1.0" encoding="UTF-8"?>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:ns=http://www.example.org/union
    targetNamespace="http://www.example.org/union">

        <xsd:element name="e21unnamed" type="ns:e21unnamed"/>

        <xsd:simpleType name="e21unnamed">
            <xsd:union>
                <xsd:simpleType>
                    <xsd:restriction base="xsd:float"/>
                </xsd:simpleType>
                <xsd:simpleType>
                    <xsd:restriction base="xsd:integer"/>
                </xsd:simpleType>
                <xsd:simpleType>
                    <xsd:restriction base="xsd:string"/>
                </xsd:simpleType>
            </xsd:union>
        </xsd:simpleType>
    </xsd:schema>
```

Will result e.g. in the following mapping:

```
module http_www_example_org_union {

import from XSD all;

type E21unnamed_1 E21unnamed
with {
    variant "name as uncapitalized";
    variant "element";
}

type union E21unnamed_1 {
    XSD.Float alt_,
    XSD.Integer alt_1,
    XSD.String alt_2,
}   with {
    variant "name as e21unnamed";
    variant "useUnion";
    variant(alt_, alt_1, alt_2) "name as ''";
    }
}
with {
    encode "XML";
    variant "namespace as 'http://www.example.org/union' prefix 'ns' ";
    variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

For instance, the below structure:

```
template E21unnamed t_UnionUnnamedInt := { alt_1 := 1 }
```

Should result e.g. in the following XML encoding:

```
<?xml version="1.0" encoding="UTF-8"?>
<ns:e21unnamed xmlns:ns='http://www.example.org/union'
    xmlns:xsd=http://www.w3.org/2001/XMLSchema
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:type='xsd:integer'>1</ns:e21unnamed>
```

EXAMPLE 3: Mixed use of named and unnamed types:

```
<xsd:simpleType name="Time-or-int-or-boolean-or-dateRestricted">
    <xsd:union memberTypes="xsd:time e21memberlist">
        <xsd:simpleType>
            <xsd:restriction base="xsd:date"/>
        </xsd:simpleType>
    </xsd:union>
</xsd:simpleType>
```

Will be mapped e.g. to the TTCN-3 type definition:

```
type union Time_or_int_or_boolean_or_dateRestricted {
    XSD.Time time,
    XSD.Integer integer_,
    XSD.Boolean boolean_,
    XSD.Date alt_
}
with {
    variant "name as 'Time-or-int-or-boolean-or-dateRestricted'";
    variant "useUnion";
    variant (integer_) "name as 'integer'";
    variant (boolean_) "name as 'boolean'";
    variant (alt_) "name as ''";
}
```

EXAMPLE 4: Mapping member type with an enumeration facet:

```
<xsd:element name="maxOccurs">
    <xsd:simpleType>
        <xsd:union memberTypes="xsd:nonNegativeInteger">
            <xsd:simpleType>
                <xsd:restriction base="xsd:token">
                    <xsd:enumeration value="unbounded"/>
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:union>
```

```
        </xsd:simpleType>
    </xsd:element>
```

Will be translated to TTCN-3 e.g. as:

```
    type union MaxOccurs {
        XSD.NonNegativeInteger nonNegativeInteger,
        enumerated {unbounded} alt_
    }
    with {
        variant "name as uncapitalized";
        variant "element";
        variant "useUnion";
        variant(alt_) "name as ''";
    }
```

EXAMPLE 5: Mapping member types with enumeration facets applied to different member types:

```
    <xsd:element name=" e22">
        <xsd:simpleType>
            <xsd:restriction base="ns:e21unnamed">
                <xsd:enumeration value="20"/>
                <xsd:enumeration value="50.0"/>
                <xsd:enumeration value="small-1"/>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:element>
```

Will be translated to TTCN-3 e.g. as:

```
    type record E22 {
      enumerated {float_, integer_, string} xsiType optional,
      enumerated {x20, x50_0, small_1} content
    }
    with {
        variant "name as uncapitalized";
        variant "element";
        variant "useUnion";
        variant (xsiType) "text 'float_' as 'float'";
        variant (xsiType) "text 'integer_' as 'integer'";
        variant (content) "text 'x20' as '20'";
        variant (content) "text 'x50_0' as '50.0'";
        variant (content) "text 'small_1' as 'small-1'";
}
```

7.6 ComplexType components

7.6.0 General

The XSD *complexType* is used for creating new types that contain elements and attributes. XSD *complexType*s may be defined globally as child of *schema* or *redefine*(in which case the *name* XSD attribute is mandatory), or locally in an anonymous fashion (as a child of *element*, without the *name* XSD attribute):

1) Translation of complexType components derived by empty extension or restriction:
   If a *complexType* is derived from another type by extension or restriction where the extension or restriction is empty, the *complexType* shall be translated to a reference to the type resulted from translating the base type of the extension or restriction. If such a type is an anonymous type of a global *element* definition, the *element* is translated to a type synonym definition (see clause 6.4 of [1]) of the type resulted from translating the *complexType*. In both cases the TTCN-3 type synonym generated shall be completed with necessary additional encoding instructions, to the base type.

2) Translation of other complexType components:
   Globally defined XSD *complexType*-s shall be translated to a TTCN-3 **`record`** type. This **`record`** type shall enframe the fields resulted by mapping the content (the children) of the XSD *complexType* as specified in the next clauses. The name of the TTCN-3 record type shall be the result of applying clause 5.2.2 to the XSD *name* attribute of the *complexType* definition.

Locally defined anonymous *complexType*s shall be ignored. In this case the **`record`** type generated for the parent *element* of the *complexType* (see clause 7.3), shall enframe the fields resulted by mapping the content (the children) of the XSD *complexType*.

NOTE: The mapping rules in subsequent clauses may be influenced by the attributes applied to the component, if any. See more details in clause 7.1, especially in clause 7.1.4.

7.6.1 ComplexType containing simple content

7.6.1.0 General

An XSD *simpleContent* component may extend or restrict an XSD simple type, being the base type of the *simpleContent* and expands the base type with attributes, but not elements.

7.6.1.1 Extending simple content

When extending XSD *simpleContent*, further XSD attributes may be added to the original type.

If the definition of a new named or unnamed complex type uses another simple or complex type as the base of the extension without changing the base type (i.e. no facet is applied and no attribute is added), it shall be translated to a TTCN-3 type synonym to the base type (see clause 6.4 of ETSI ES 201 873-1 [1]), completed with necessary additional encoding instructions (see clause 7.6 rule 1).

NOTE: This means that tools need not analyse the effective value space of the base and the derived types, but can make a decision based on the presence of facet(s) and XSD *attribute* elements in the derived type.

EXAMPLE 1: Examples of extensions without effectively changing the base XSD type:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.example.org/aliases-ext"
    xmlns:ns="http://www.example.org/aliases-ext">

    <xsd:complexType name="complex-base-simple">
        <xsd:simpleContent>
            <xsd:extension base="xsd:integer"/>
        </xsd:simpleContent>
    </xsd:complexType>

    <xsd:complexType name="complex-ext-simple">
        <xsd:simpleContent>
            <xsd:extension base="ns:complex-base-simple"/>
        </xsd:simpleContent>
    </xsd:complexType>

    <xsd:element name="elem-complex-ext-simple">
        <xsd:complexType>
            <xsd:simpleContent>
                <xsd:extension base="ns:complex-base-simple"/>
            </xsd:simpleContent>
        </xsd:complexType>
    </xsd:element>

</xsd:schema>
```

Will be translated to the TTCN-3 module, e.g. as:

```
module http_www_example_org_aliases_ext {

    import from XSD all;

    type XSD.Integer Complex_base_simple
    with {
      variant "name as 'complex-base-simple'"
    };

    type Complex_base_simple Complex_ext_simple
    with {
      variant "name as 'complex-ext-simple'"
    };
```

```
    type Complex_base_simple Elem_complex_ext_simple
    with {
      variant "name as 'elem-complex-ext-simple'";
      variant "element";
    };
}
with {
  encode "XML";
  variant "namespace as 'http://www.example.org/aliases-ext' prefix 'ns'";
  variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

In all other cases the base type of the extended *simpleContent* and the additional XSD attributes shall be mapped to fields of the TTCN-3 **record** type, generated for the enclosing XSD complexType (see clause 7.6). At first, attribute elements and attribute groups shall be translated according to clause 7.6.7, and added to the enframing TTCN-3 **record** (see clause 7.6). Next, the extended type shall be mapped to TTCN-3 and added as a field of the enframing **record**. The field name of the latter shall be "**base**" and the variant attribute "untagged" shall be attached to it.

EXAMPLE 2: The example below extends a built-in type by adding an attribute:

```
<xsd:element name="e23">
    <xsd:complexType>
        <xsd:simpleContent>
            <xsd:extension base="string">
                <xsd:attribute name="foo" type="xsd:float"/>
                <xsd:attribute name="bar" type="xsd:integer"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>
</xsd:element>
```

Will be mapped to TTCN-3 e.g. as:

```
type record E23
{
    XSD.Integer bar optional,
    XSD.Float foo optional,
    XSD.String base
}
with {
    variant "name as uncapitalized";
    variant "element";
    variant (base) "untagged";
    variant (bar, foo) "attribute";
}
```

and the template:

```
template E23 t_E23 := {
  bar := 1,
  foo := 2.0,
  base := "something"
}
```

Will be encoded in XML e.g. as:

```
<?xml version="1.0" encoding="UTF-8"?>
<e23 bar="1" foo="2.0">something</e23>
```

7.6.1.2 Restricting simple content

If the definition of a new named or unnamed complex type uses another simple or complex type as the base of the restriction without changing the base type (i.e. no facet is applied), it shall be translated to a TTCN-3 type synonym to the base type (see clause 6.4 of ETSI ES 201 873-1 [1]), completed with necessary additional encoding instructions (see clause 7.6 rule 1).

NOTE: This means that tools need not analyse the effective value space of the base and the derived types, but can make a decision based on the presence of facet(s) in the derived type.

EXAMPLE 1: The example below extends a built-in type by adding an attribute:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.example.org/aliases-restr"
    xmlns:ns="http://www.example.org/aliases-restr">

    <xsd:complexType name="complex-base-simple">
        <xsd:simpleContent>
            <xsd:extension base="xsd:integer"/>
        </xsd:simpleContent>
    </xsd:complexType>

    <xsd:complexType name="complex-restr-simple">
        <xsd:simpleContent>
            <xsd:restriction base="ns:complex-base-simple"/>
        </xsd:simpleContent>
    </xsd:complexType>

    <xsd:element name="elem-complex-restr-simple">
        <xsd:complexType>
            <xsd:simpleContent>
                <xsd:restriction base="ns:complex-base-simple"/>
            </xsd:simpleContent>
        </xsd:complexType>
    </xsd:element>

</xsd:schema>
```

Will be translated to the TTCN-3 module, e.g. as:

```
module http_www_example_org_aliases_restr {

    type XSD.Integer Complex_base_simple
    with {
      variant "name as 'complex-base-simple'"
    };

    type Complex_base_simple Complex_restr_simple
    with {
      variant "name as 'complex-restr-simple'";
    };

    type Complex_base_simple Elem_complex_restr_simple
    with {
      variant "name as 'elem-complex-restr-simple'";
      variant "element";
    };

}
    with {
      encode "XML";
      variant "namespace as 'http://www.example.org/aliases-restr' prefix 'ns'";
      variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
    }
```

An XSD *simpleContent* may restrict its base type or attributes of the base type by applying more restrictive facets than those of the base type (if any). Such XSD *simpleContent* shall be mapped to fields of the enframing TTCN-3 **record** (see clause 7.6). At first, the fields corresponding to the local attribute definitions, attribute and attributeGroup references shall be generated according to clause 7.6.7, followed by the field generated for the base type. The field name of the latter shall be "base". The restrictions of the given *simpleContent* shall be applied to the "base" field directly (i.e. the base type shall not be referenced but translated to a new type definition in TTCN-3).

Other base types shall be dealt with accordingly, see clause 6.

EXAMPLE 2: Example for restriction of a base type:

```
<xsd:element name="e24">
    <xsd:complexType>
        <xsd:simpleContent>
            <xsd:restriction base="ns:e23">
                <xsd:length value="4"/>
            </xsd:restriction>
        </xsd:simpleContent>
    </xsd:complexType>
</xsd:element>
```

Will be translated to TTCN-3 e.g. as:

```
type record E24 {
    XSD.Integer bar optional,
    XSD.Float foo optional,
    XSD.String base length(4)
}
with {
    variant "name as uncapitalized";
    variant "element";
    variant(base) "untagged";
    variant(bar, foo) "attribute";

}
```

And the template:

```
template E24 t_E24 := {
  bar := 1,
  foo := 2.0,
  base := "some"
}
```

Can be encoded in XML, for example, as:

```
<?xml version="1.0" encoding="UTF-8"?><e24 bar="1" foo="2.0">some</e24>
```

7.6.2 ComplexType containing complex content

7.6.2.0 General

In contrast to *simpleContent*, *complexContent* is allowed to have elements. It is possible to extend a base type by adding attributes or elements, it is also possible to restrict a base type to certain elements or attributes.

7.6.2.1 Complex content derived by extension

By using the XSD *extension* for a *complexContent* it is possible to derive new complex types from a base (complex) type by adding attributes, elements or groups (*group*, *attributeGroup*). The compositor of the base type may be *sequence* or *choice* (i.e. complex types with the compositor *all* shall not be extended).

If the definition of a new named or unnamed complex type uses another simple or complex type as the base of the extension without changing the base type (i.e. no facet is applied and no *element* or *attribute* is added), it shall be translated to a TTCN-3 type synonym to the base type (see clause 6.4 of ETSI ES 201 873-1 [1]), completed with necessary additional encoding instructions (see clause 7.6 rule 1).

NOTE: This means that tools need not analyse the effective value space of the base and the derived types, but can make a decision based on the presence of facet(s) and XSD *attribute* or *element* elements in the derived type.

EXAMPLE 1: The example below extends a complex type without effectively changing it:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.example.org/aliases-ccext"
    xmlns:ns="http://www.example.org/aliases-ccext">

    <xsd:complexType name="complex-base-complex">
        <xsd:sequence>
            <xsd:element name="int" type="xsd:integer"/>
            <xsd:element name="str" type="xsd:string"/>
        </xsd:sequence>
        <xsd:attribute name="attr" type="xsd:integer"/>
    </xsd:complexType>

    <xsd:complexType name="complex-ext-complex">
        <xsd:complexContent>
            <xsd:extension base="ns:complex-base-complex" />
        </xsd:complexContent>
    </xsd:complexType>
```

```
    <xsd:element name="elem-complex-ext-complex">
        <xsd:complexType>
            <xsd:complexContent>
                <xsd:extension base="ns:complex-base-complex" />
        </xsd:complexContent>
        </xsd:complexType>
    </xsd:element>

</xsd:schema>
```

Will be translated to the TTCN-3 module, e.g. as:

```
module http_www_example_org_aliases_ccext {

    type record Complex_base_complex
    {
        XSD.Integer attr optional,
        XSD.Integer int,
        XSD.String str
    }
    with {
      variant "name as 'complex-base-complex'";
      variant (attr) "attribute";
    };

    type Complex_base_complex Complex_ext_complex
    with {
      variant "name as 'complex-ext-complex'";
    };

    type Complex_base_complex Elem_complex_ext_complex
    with {
      variant "name as 'elem-complex-ext-complex'";
      variant "element";
    };
}
with {
  encode "XML";
  variant "namespace as 'http://www.example.org/aliases-ccext' prefix 'ns'";
  variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

All other cases shall be translated to TTCN-3 as follows (the generated TTCN-3 constructs shall be added to the enframing TTCN-3 **`record,`** see clause 7.6, in the order of the items below):

a) At first, attributes and attribute and attribute group references of the base type and the extending type shall be translated according to clause 7.6.7 and the resulted fields added to the enframing TTCN-3 **`record`** directly (i.e. without nesting).

b) The *choice* or *sequence* content model of the base (extended) *complexType* shall be mapped to TTCN-3 according to clauses 7.6.5 or 7.6.6 respectively, and the resulted TTCN-3 constructs shall be added to the enframing **`record`**.

c) The extending *choice* or *sequence* content model of the extending *complexContent* shall be mapped to TTCN-3 according to clauses 7.6.5 or 7.6.6 respectively, and the resulted TTCN-3 constructs shall be added to the enframing **`record`**.

EXAMPLE 2: Both the base and the extending types have the compositor *sequence*:

The base definitions:

```
<xsd:complexType name="e25seq">
    <xsd:sequence>
        <xsd:element name="titleElemBase" type="xsd:string"/>
        <xsd:element name="forenameElemBase" type="xsd:string"/>
        <xsd:element name="surnameElemBase" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="genderAttrBase" type="xsd:integer"/>
    <xsd:attributeGroup ref="ns:g25attr2"/>
</xsd:complexType>

<xsd:group name="g25seq">
    <xsd:sequence>
```

```
        <xsd:element name="familyStatusElemInGroup" type="xsd:string"/>
        <xsd:element name="spouseElemInGroup" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
</xsd:group>

<xsd:attributeGroup name="g25attr1">
    <xsd:attribute name="birthPlaceAttrGroup" type="xsd:string"/>
    <xsd:attribute name="birthDateAttrGroup" type="xsd:string"/>
</xsd:attributeGroup>

<xsd:attributeGroup name="g25attr2">
    <xsd:attribute name="jobPositionAttrGroup" type="xsd:string"/>
</xsd:attributeGroup>
```

Now a type is defined that extends e25seq by adding a new element, group and attributes:

```
<xsd:complexType name="e26seq">
    <xsd:complexContent>
        <xsd:extension base="ns:e25seq">
            <xsd:sequence>
                <xsd:element name="ageElemExt" type="xsd:integer"/>
                <xsd:group ref="ns:g25seq"/>
            </xsd:sequence>
            <xsd:attribute name="unitOfAge" type="xsd:string"/>
            <xsd:attributeGroup ref="ns:g25attr1"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

This is translated to the TTCN-3 structure, e.g. as:

```
type record E26seq
{
    // fields corresponding to attributes of the base and the extending type
    // (in alphabetical order)
    XSD.String birthDateAttrGroup optional,
    XSD.String birthPlaceAttrGroup optional,
    XSD.Integer genderAttrBase optional,
    XSD.String jobPositionAttrGroup optional,
    XSD.String unitOfAge optional,
    // followed by fields corresponding to elements of the base type
    XSD.String titleElemBase,
    XSD.String forenameElemBase,
    XSD.String surnameElemBase,
    // finally fields corresponding to the extending element and group reference
    XSD.Integer ageElemExt,
    G25seq g25seq
}
with {
    variant "name as uncapitalized ";
    variant (birthDateAttrGroup, birthPlaceAttrGroup, genderAttrBase, jobPositionAttrGroup,
            unitOfAge) "attribute";
};
```

where:

```
type record G25seq {
    XSD.String  familyStatusElemInGroup,
    XSD.String  spouseElemInGroup       optional
}
with {
    variant "untagged";
}
```

EXAMPLE 3: Both the base and the extending types have the compositor *sequence* and multiple occurrences are allowed:

Additional base definition:

```
<xsd:complexType name="e25seqRecurrence">
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="titleElemBase" type="xsd:string"/>
        <xsd:element name="forenameElemBase" type="xsd:string"/>
        <xsd:element name="surnameElemBase" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="genderAttrBase" type="xsd:integer"/>
```

```
    <xsd:attributeGroup ref="ns:g25attr2"/>
</xsd:complexType>
```

The extending type definition:

```
<xsd:complexType name="e26seqReccurrence">
    <xsd:complexContent>
        <xsd:extension base="ns:e25seq">
            <xsd:sequence minOccurs="0" maxOccurs="unbounded">
                <xsd:group ref="ns:g25seq"/>
                <xsd:element name="ageElemExt" type="xsd:integer"/>
            </xsd:sequence>
            <xsd:attribute name="unitOfAge" type="xsd:string"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="e26seqDoubleRecurrence">
    <xsd:complexContent>
        <xsd:extension base="ns:e25seqRecurrence">
            <xsd:sequence minOccurs="0" maxOccurs="unbounded">
                <xsd:group ref="ns:g25seq"/>
                <xsd:element name="ageElemExt" type="xsd:integer"/>
            </xsd:sequence>
            <xsd:attribute name="unitOfAge" type="xsd:string"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

The extending types are translated to TTCN-3 e.g. as:

```
type record E26seqRecurrence {
    // fields corresponding to attributes of the base and the extending type
    // (in alphabetical order)
    XSD.Integer genderAttrBase optional,
    XSD.String jobPositionAttrGroup optional,
    XSD.String unitOfAge optional,
    // followed by a "simple" field list corresponding to elements of the base type
    XSD.String titleElemBase,
    XSD.String forenameElemBase,
    XSD.String surnameElemBase,
    // the extending sequence is recurring (see clause 7.6.6.6 for the mapping)
    record of record {
        G25seq g25seq,
        XSD.Integer ageElemExt,
    } sequence_list
}
with {
    variant "name as uncapitalized";
    variant(sequence_list) "untagged";
    variant (genderAttrBase, jobPositionAttrGroup, unitOfAge) "attribute";
}

type record E26seqDoubleRecurrence {
    // fields corresponding to attributes of the base and the extending type
    // (in alphabetical order)
    XSD.Integer genderAttrBase optional,
    XSD.String jobPositionAttrGroup optional,
    XSD.String unitOfAge optional,
    // followed by a record of record field containing the fields corresponding to elements of
    // the base type; the base type is a recurring sequence (see clause
    // 7.6.6.6 for the
    // mapping)
    record of record {
        XSD.String titleElemBase,
        XSD.String forenameElemBase,
        XSD.String surnameElemBase
    } sequence_list,
    // the extending sequence is recurring too(see clause
    // 7.6.6.6 for the
    // mapping)
    record of record {
        G25seq g25seq,
        XSD.Integer ageElemExt,
    } sequence_list_1
}
with {
    variant "name as uncapitalized";
```

```
    variant(sequence_list, sequence_list_1) "untagged";
    variant (genderAttrBase, jobPositionAttrGroup, unitOfAge) "attribute";
}
```

EXAMPLE 4: Both the base and the extending types have the compositor *choice*:

```
<xsd:complexType name="e25cho">
    <xsd:choice>
        <xsd:element name="titleElemBase" type="xsd:string"/>
        <xsd:element name="forenameElemBase" type="xsd:string"/>
        <xsd:element name="surnameElemBase" type="xsd:string"/>
    </xsd:choice>
    <xsd:attribute name="genderAttrBase" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="e26cho">
    <xsd:complexContent>
        <xsd:extension base="ns:e25cho">
            <xsd:choice>
                <xsd:element name="ageElemExt" type="xsd:integer"/>
                <xsd:element name="birthdayElemExt" type="xsd:date"/>
            </xsd:choice>
            <xsd:attribute name="unitAttrExt" type="xsd:string"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Are translated to TTCN-3 e.g. as:

```
type record E26cho {
    XSD.String genderAttrBase optional,
    XSD.String unitAttrExt optional,
    union {
        XSD.String titleElemBase,
        XSD.String forenameElemBase,
        XSD.String surnameElemBase
    } choice,
    union {
        XSD.Integer ageElemExt,
        XSD.Date birthdayElemExt
    } choice_1
}
with {
    variant "name as uncapitalized";
    variant(genderAttrBase, unitAttrExt) "attribute";
    variant(choice, choice_1) "untagged";
}
```

EXAMPLE 5: Extension of a *sequence* base type by a *choice* model group:

```
<xsd:complexType name="e27cho">
    <xsd:complexContent>
        <xsd:extension base="ns:e25seq">
            <xsd:choice>
                <xsd:element name="ageElemExt" type="xsd:integer"/>
                <xsd:element name="birthdayElemExt" type="xsd:date"/>
            </xsd:choice>
            <xsd:attribute name="unitAttrExt" type="xsd:string"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record E27cho
{
    XSD.Integer genderAttrBase optional,
    XSD.String jobPositionAttrGroup optional,
    XSD.String unitAttrExt optional,
    XSD.String titleElemBase,
    XSD.String forenameElemBase,
    XSD.String surnameElemBase,
    union {
        XSD.Integer ageElemExt,
        XSD.Date birthdayElemExt
    } choice
```

```
}
with {
    variant "name as uncapitalized";
    variant(genderAttrBase, jobPositionAttrGroup, unitAttrExt) "attribute";
    variant(choice) "untagged";
}
```

EXAMPLE 6: Extending of a base type with *choice* model group by a *sequence* model group:

```
<xsd:complexType name="e27seq">
    <xsd:complexContent>
        <xsd:extension base="ns:e25cho">
            <xsd:sequence>
                <xsd:element name="ageElemExt" type="xsd:integer"/>
            </xsd:sequence>
            <xsd:attribute name="unitAttrExt" type="xsd:string"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record E27seq {
    XSD.String genderAttrBase optional,
    XSD.String unitAttrExt optional,
    union {
        XSD.String titleElemBase,
        XSD.String forenameElemBase,
        XSD.String surnameElemBase
    } choice,
    XSD.Integer ageElemExt
}
with {
    variant "name as uncapitalized";
    variant(genderAttrBase, unitAttrExt) "attribute";
    variant(choice) "untagged";
}
```

EXAMPLE 7: Recursive extension of an anonymous inner type is realized using the TTCN-3 dot notation (starts from the name of the outmost type):

```
<xsd:complexType name="X">
    <xsd:sequence>
        <xsd:element name="x" type="xsd:string"/>
        <xsd:element name="y" minOccurs="0">
            <xsd:complexType>
                <xsd:complexContent>
                    <xsd:extension base="ns:X">
                        <xsd:sequence>
                            <xsd:element name="z" type="xsd:string"/>
                        </xsd:sequence>
                    </xsd:extension>
                </xsd:complexContent>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>
```

Will be translated to the TTCN-3 structure e.g. as:

```
type record X {
    XSD.String x,
    record {
        XSD.String x,
        X.y y optional,
        XSD.String z
    } y optional
}
```

7.6.2.2 Complex content derived by restriction

The *restriction* uses a base complex type and allows to restrict one or more of its components.

If the definition of a new named or unnamed complex type uses another complex type as the base of the restriction without changing the base type (i.e. no facet is present), it shall be translated to a TTCN-3 type synonym to the base type (see clause 6.4 of ETSI ES 201 873-1 [1]), completed with necessary additional encoding instructions (see clause 7.6 rule 1).

NOTE: This means that tools need not analyse the effective value space of the base and the derived types, but can make a decision based on the presence of facet(s) in the derived type.

EXAMPLE 1: The example below restricts a base complex type without effectively changing it:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.example.org/aliases-ccrestr"
    xmlns:ns="http://www.example.org/aliases-ccrestr">

    <xsd:complexType name="complex-base-empty"/>

    <xsd:complexType name="complex-restr-complex">
        <xsd:complexContent>
            <xsd:restriction base="ns:complex-base-empty" />
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:element name="elem-complex-restr-complex">
            <xsd:complexType>
                <xsd:complexContent>
                    <xsd:restriction base="ns:complex-base-empty" />
            </xsd:complexContent>
            </xsd:complexType>
    </xsd:element>

</xsd:schema>
```

Will be translated to the TTCN-3 module, e.g. as:

```
module http_www_example_org_aliases_ccrestr {

    type record Complex_base_empty { }
    with {
      variant "name as 'complex-base-empty'";
    };

    type Complex_base_empty Complex_restr_complex
    with {
      variant "name as 'complex-restr-complex'";
    };

    type Complex_base_empty Elem_complex_restr_complex
    with {
      variant "name as 'elem-complex-restr-complex'";
      variant "element";
    };

}
with {
  encode "XML";
  variant "namespace as 'http://www.example.org/aliases-ccrestr' prefix 'ns'";
  variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

In all other cases all components present in the restricted type shall be mapped to TTCN-3, applying the restrictions, and the resulted fields shall be added to the enframing TTCN-3 **`record`** (see clause 7.6). Thus neither the base type nor its components are referenced from the restricted type.

EXAMPLE 2: Restricting *anyType*: in the example below *anyType* (any possible type) is used as the base type and it is restricted to only two elements:

```
<xsd:complexType name="e28">
    <xsd:complexContent>
        <xsd:restriction base="anyType">
            <xsd:sequence>
                <xsd:element name="size" type="nonPositiveInteger"/>
                <xsd:element name="unit" type="NMTOKEN"/>
            </xsd:sequence>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record E28 {
    XSD.NonPositiveInteger size,
    XSD.NMTOKEN unit
}
with {
    variant "name as uncapitalized";
}
```

EXAMPLE 3: Restricting a user defined complex type (the effect of the *use* attribute is described in clause 7.1.12):

```
<xsd:element name="comment" type="xsd:string"/>
```

The base type is:

```
<xsd:complexType name="PurchaseOrderType">
    <xsd:sequence>
        <xsd:element name="shipTo"     type="xsd:string"/>
        <xsd:element name="billTo"     type="xsd:string"/>
        <xsd:element ref="ns:comment" minOccurs="0"/>
        <xsd:element name="items"      type="ns:Items"/>
    </xsd:sequence>
    <xsd:attribute name="shipDate" type="xsd:date"/>
    <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>
```

The restricting type is:

```
<xsd:complexType name="RestrictedPurchaseOrderType">
    <xsd:complexContent>
        <xsd:restriction base="ns:PurchaseOrderType">
            <xsd:sequence>
                <xsd:element name="shipTo" type="xsd:string"/>
                <xsd:element name="billTo" type="xsd:string"/>
                <xsd:element ref="ns:comment" minOccurs="1"/>
                <xsd:element name="items"  type="ns:Items"/>
            </xsd:sequence>
            <xsd:attribute name="shipDate" type="xsd:date" use="required" />
            <xsd:attribute name="orderDate" type="xsd:date" use="prohibited" />
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type XSD.String Comment
with {
    variant "name as uncapitalized";
    variant "element";
}

type record PurchaseOrderType {
    XSD.Date orderDate optional,
    XSD.Date shipDate optional,
    XSD.String shipTo,
    XSD.String billTo,
    Comment comment optional,
    Items items
}
with {
```

```
    variant (orderDate, shipDate) "attribute";
}

type record RestrictedPurchaseOrderType {
    XSD.Date shipDate,  //note that this field become mandatory
                              //note that the field orderDate is not added
    XSD.String shipTo,
    XSD.String billTo,
    Comment comment,    //note that this field become mandatory
    Items items
}
with {
    variant (orderDate) "attribute";
}
```

7.6.3 Referencing group components

Referenced model *group* components shall be translated as follows:

- when *group* reference is a child of *complexType*, the compositor of the referenced group definition is *sequence* and both the *minOccurs* and *maxOccurs* attributes of the group reference equal to "1" (either explicitly or by defaulting to "1"), it shall be translated as if the child *element*s of the referenced group definition were present in the *complexType* definition directly;

- when the referenced *group* has the compositor *all*, it has to be translated is the content of the referenced group definition was present directly, i.e. according to clause 7.6.4;

- in all other cases the referenced group component shall be translated to a field of the enclosing record of type (generated for the parent *complexType*, *sequence* or *choice* element) referencing the TTCN-3 type generated for the referenced *group* definition, considering also the attributes of the referenced group component according to clause 7.1.

NOTE: Please note, as the "untagged" attribute is applied to the TTCN-3 type generated for the referenced model group, the name of the field corresponding to the group reference will never appear in an encoded XML value.

When a referenced g*roup* is defined in an XSD Schema with a target namespace, different from the target namespace of the referencing XSD schema (including the no target namespace case), all TTCN-3 fields generated for this *group* reference shall be appended with a "namespace as" encoding instruction (see clause B.3.1), which shall identify the namespace and optionally the prefix of the XSD schema in which the referenced entity is defined.

EXAMPLE 1: Mapping of a group reference, child of *complexType*, compositor *<xsd:sequence>*:

Referencing a group with compositor <sequence> (see group declaration in clause 7.9):

```
<xsd:complexType name="LonelySeqGroup">
    <xsd:group ref="ns:shipAndBill"/>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record LonelySeqGroup {
    XSD.String shipTo,
    XSD.String billTo
}
```

The group reference is optional, compositor <sequence> (see group declaration in clause 7.9):

```
<xsd:complexType name="LonelySeqGroupOptional">
    <xsd:group ref="ns:shipAndBill" minOccurs="0"/>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record LonelySeqGroupOptional {
    ShipAndBill shipAndBill optional
}
```

The group reference is iterative, compositor <sequence> (see group declaration in clause 7.9):

```
<xsd:complexType name="LonelySeqGroupRecurrence">
        <xsd:group ref="ns:shipAndBill" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record LonelySeqGroupRecurrence {
    record of ShipAndBill shipAndBill_list
}
with {
    variant (shipAndBill_list) "untagged";
}
```

Referencing a group with compositor <all> (see group declaration in clause 7.9):

```
<xsd:complexType name="LonelyAllGroup">
    <xsd:group ref="ns:shipAndBillAll"/>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record LonelyAllGroup {
    record of enumerated { shipTo, billTo } order,
    XSD.String shipTo,
    XSD.String billTo
}
with {
    variant "useOrder";
}
```

The group reference is optional, compositor <all> (see group declaration in clause 7.9):

```
<xsd:complexType name="LonelyAllGroupOptional">
    <xsd:group ref="ns:shipAndBillAll" minOccurs="0"/>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record LonelyAllGroupOptional {
    record of enumerated { shipTo, billTo } order,
    XSD.String shipTo optional,
    XSD.String billTo optional
}
with {
    variant "useOrder";
}
```

EXAMPLE 2: Mapping of a group reference, child of *complexType*, compositor *<xsd:choice>*:

Referencing a group with compositor <choice> (see group declaration in clause 7.9):

```
<xsd:complexType name="LonelyChoGroup">
    <xsd:group ref="ns:shipOrBill"/>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record LonelyChoGroup {
    ShipOrBill shipOrBill
}
```

The group reference is optional, compositor <choice> (see group declaration in clause 7.9):

```
<xsd:complexType name="LonelyChoGroupOptional">
    <xsd:group ref="ns:shipOrBill" minOccurs="0"/>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record LonelyChoGroupOptional {
    ShipOrBill shipOrBill optional
}
```

```
<xsd:complexType name="LonelyChoGroupRecurrence">
<annotation><documentation xml:lang="EN">choice group reference</documentation></annotation>
    <xsd:group ref="ns:shipOrBill" minOccurs="0" maxOccurs="unbounded"/>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record LonelyChoGroupRecurrence {
    //choice group reference
    record of ShipOrBill shipOrBill_list
}
with {
    variant (shipOrBill_list) "untagged";
}
```

EXAMPLE 3: Mapping of group references, children of <sequence> or <choice>:

Referencing a group with compositor <sequence> in <sequence> (see group declaration in clause 7.9):

```
<xsd:complexType name="SeqGroupAndElementsInSequence">
    <xsd:sequence id="embeddingSequence">
        <xsd:group ref="ns:shipAndBill"/>
        <xsd:element name="comment" type="xsd:string" minOccurs="0" />
        <xsd:element name="items"  type="xsd:string" />
    </xsd:sequence>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record SeqGroupAndElementsInSequence {
    ShipAndBill shipAndBill,
    XSD.String comment optional,
    XSD.String items
}
```

Referencing a group with compositor <sequence> in <choice> (see group declaration in clause 7.9):

```
<xsd:complexType name="SeqGroupAndElementsAndAttributeInChoice">
    <xsd:choice id="embeddingChoice">
        <xsd:annotation>
            <xsd:documentation xml:lang="EN">sequence group ref.</xsd:documentation>
        </xsd:annotation>
        <xsd:group ref="ns:shipAndBill"/>
        <xsd:element name="comment" minOccurs="0" type="xsd:string"/>
        <xsd:element name="items"  type="xsd:string"/>
    </xsd:choice>
    <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record SeqGroupAndElementsAndAttributeInChoice {
    XSD.Date orderDate optional,
    union {
    /* sequence group ref.*/
    ShipAndBill shipAndBill,
    record length (0..1) of XSD.String comment_list,
    XSD.String items
    } choice
}
with {
    variant (orderDate) "attribute";
    variant (choice) "untagged";
    variant (choice.comment_list) "untagged";
    variant (choice.comment_list[-]) "name as comment";
}
```

7.6.4 All content

An XSD *all* compositor defines a collection of elements, which can appear in any order in an XML value.

In the general case, when the values of both the *minOccurs* and *maxOccurs* attributes of the *all* compositor equal "1" (either explicitly or by defaulting to "1"), it shall be translated to TTCN-3 by adding the fields resulted by mapping the XSD elements to the enframing TTCN-3 **`record`** (see clause 7.6). By setting the *minOccurs* XSD attribute of the *all* compositor to 0, all elements of the *all* content model are becoming optional. In this case all record fields corresponding to the elements of the *all* model group shall be set to **`optional`** too. In addition, to these fields, an extra first field named "order" shall be inserted into the enframing **`record`**. The type of this extra field shall be **`record of enumerated`**, where the names of the enumeration values shall be the names of the fields resulted by mapping the elements of the *all* structure. Finally, a "useOrder" variant attribute shall be attached to the enframing **`record`**.

The `order` field shall precede the fields resulted by the translation of the *attribute*s and attribute and *attributeGroup* references of the given complexType but shall follow the `embed_values` field, if any, generated for the *mixed*="true" attribute value (see also clause 7.6.8).

NOTE: When encoding, the presence and order of elements in the encoded XML instance will be controlled by the `order` field. This is indicated by the "useOrder" encoding instruction. When decoding, the presence and order of elements in the XML instance will control the value of the `order` field that appears in the decoded structure. See more details in annex B. This mapping is required by the alignment to Recommendation ITU-T X.694 [4].

EXAMPLE 1: XSD *all* content model with mandatory elements:

```
<xsd:complexType name="e29a">
    <xsd:all>
        <xsd:element name="foo" type="xsd:integer"/>
        <xsd:element name="bar" type="xsd:float"/>
        <xsd:element name="ding" type="xsd:string"/>
    </xsd:all>
</xsd:complexType>
```

Will be mapped to the TTCN-3 structure e.g. as:

```
type record E29a {
    record of enumerated {foo,bar,ding} order,
    XSD.Integer foo,
    XSD.Float  bar,
    XSD.String ding
}
with {
    variant "name  as uncapitalized ";
    variant "useOrder";
}
```

EXAMPLE 2: XSD *all* content model with each element being optional:

```
<xsd:complexType name="e29b">
    <xsd:all minOccurs="0">
        <xsd:element name="foo" type="xsd:integer"/>
        <xsd:element name="bar" type="xsd:float"/>
        <xsd:element name="ding" type="xsd:string"/>
    </xsd:all>
</xsd:complexType>
```

Will be mapped to the TTCN-3 structure e.g. as:

```
type record E29b {
  record of enumerated {foo,bar,ding} order,
    XSD.Integer foo optional,
    XSD.Float  bar optional,
    XSD.String ding optional
}
with {
    variant "name as uncapitalized ";
    variant "useOrder";
}
```

EXAMPLE 3: XSD *all* content model, with selected optional elements:

```
<xsd:complexType name="e29c">
    <xsd:all>
        <xsd:element name="foo" type="xsd:integer"/>
        <xsd:element name="bar" type="xsd:float" minOccurs="0"/>
        <xsd:element name="ding" type="xsd:string"/>
    </xsd:all>
</xsd:complexType>
```

Will be mapped to the TTCN-3 structure e.g. as:

```
type record E29c {
    record of enumerated {foo,bar,ding} order,
    XSD.Integer foo,
    XSD.Float  bar optional,
    XSD.String ding
}
with {
    variant "name as uncapitalized ";
    variant "useOrder";
}
```

EXAMPLE 4: XSD complex type with attributes and *all* content model:

```
<xsd:attribute name="attrGlobal" type="token"/>

<xsd:attributeGroup name="attrGroup">
    <xsd:attribute name="attrInGroup2" type="token"/>
    <xsd:attribute name="attrInGroup1" type="token"/>
</xsd:attributeGroup>

<xsd:complexType name="e29aAndAttributes">
    <xsd:all>
        <xsd:element name="foo" type="xsd:integer"/>
        <xsd:element name="bar" type="xsd:float"/>
        <xsd:element name="ding" type="xsd:string"/>
    </xsd:all>
    <xsd:attribute name="attrLocal" type="xsd:integer"/>
    <xsd:attribute ref="ns:attrGlobal"/>
    <xsd:attributeGroup ref="ns:attrGroup"/>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record E29aAndAttributes {
    record of enumerated { foo, bar, ding } order,
    XSD.Token attrInGroup1 optional,
    XSD.Token attrInGroup2 optional,
    XSD.Integer attrLocal optional,
    XSD.Token attrGlobal optional,
    XSD.Integer foo,
    XSD.Float bar,
    XSD.String ding
}
with {
    variant "name as uncapitalized";
    variant "useOrder";
    variant(attrInGroup1, attrInGroup2, attrLocal, attrGlobal) "attribute";
}
```

7.6.5 Choice content

7.6.5.0 General

An XSD *choice* content defines a collection of mutually exclusive alternatives.

In the general case, when both the *minOccurs* and *maxOccurs* attribute equal to "1" (either explicitly or by defaulting to "1"), it shall be mapped to a TTCN-3 **union** field with the field name "choice" and the encoding instruction "untagged" shall be attached to this field.

If the value of the *minOccurs* or the *maxOccurs* attributes or both differ from "1", the following rules shall apply:

a) The union field shall be generated as above (including attaching the "untagged" encoding instruction).

b) The procedures in clause 7.1.4 shall be called for the **`union`** field.

NOTE: As the result of applying clause 7.1.4, the type of the field may be changed to **`record of union`** and in parallel the name of the field should be changed to "choice_list".

c) Finally, clause 5.2.2 shall be applied to the name of the resulted field and subsequently the field shall be added to the enframing TTCN-3 record type (see clause 7.6) or record or union field corresponding to the parent of the mapped *choice* compositor.

The content for a choice component may be any combination of *element*, *group*, *choice*, *sequence* or *any*. The following clauses discuss the mapping for various contents nested in a choice component.

7.6.5.1 Choice with nested elements

Nested elements shall be mapped as fields of the enframing TTCN-3 **`union`** or **`record of union`** field (see clause 7.6.5) according to clause 7.3.

EXAMPLE:

```
<xsd:complexType name="e30">
    <xsd:choice>
        <xsd:element name="foo" type="xsd:integer"/>
        <xsd:element name="bar" type="xsd:float"/>
    </xsd:choice>
</xsd:complexType>
```

Will be translated toTTCN-3 e.g. as:

```
type record E30 {
    union {
        XSD.Integer foo,
        XSD.Float  bar
    } choice
}
with {
    variant "name as uncapitalized";
    variant(choice) "untagged";
}
```

7.6.5.2 Choice with nested group

Nested group components shall be mapped along with other content as a field of the enframing TTCN-3 union or **`record of union`** field (see clause 7.6.5). The type of this field shall refer to the TTCN-3 type generated for the corresponding group and the name of the field shall be the name of the TTCN-3 type with the first character uncapitalized.

EXAMPLE: The following example shows this with a *sequence* group and an *element*:

```
<xsd:group name="e31">
    <xsd:sequence>
        <xsd:element name="foo" type="xsd:string"/>
        <xsd:element name="bar" type="xsd:string"/>
    </xsd:sequence>
</xsd:group>

<xsd:complexType name="e32">
    <xsd:choice>
        <xsd:group ref="ns:e31"/>
        <xsd:element name="ding" type="xsd:string"/>
    </xsd:choice>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record E31 {
    XSD.String foo,
    XSD.String bar
}
with
{
    variant "name as uncapitalized ";
}

type record E32 {
    union {
        E31 e31,
        XSD.String ding
    } choice
}
with {
    variant "name as uncapitalized ";
    variant(choice) "untagged";
}
```

7.6.5.3 Choice with nested choice

An XSD *choice* nested to a *choice* shall be translated according to clause 7.6.5:

EXAMPLE:

```
<xsd:complexType name="e33">
    <xsd:choice>
        <xsd:choice>
            <xsd:element name="foo" type="xsd:string"/>
            <xsd:element name="bar" type="xsd:string"/>
        </xsd:choice>
        <xsd:element name="ding" type="xsd:string"/>
    </xsd:choice>
</xsd:complexType>
```

Will be mapped to TTCN-3 e.g. as:

```
type record E33 {
    union {
        union {
            XSD.String foo,
            XSD.String bar
        } choice,
        XSD.String ding
    } choice
}
with {
    variant "name as uncapitalized";
    variant(choice, choice.choice) "untagged";
}
```

7.6.5.4 Choice with nested sequence

An XSD *sequence* nested to a *choice* shall be mapped to a TTCN-3 **`record`** field of the enframing TTCN-3 **`union`** or **`record of union`** field (see clause 7.6.5), according to clause 7.6.6. The name of the **`record`** field shall be the result of applying clause 5.2.2 to "sequence".

EXAMPLE 1: Single *sequence* nested to *choice*:

```
<xsd:complexType name="e34a">
    <xsd:choice>
        <xsd:sequence>
            <xsd:element name="foo" type="xsd:string"/>
            <xsd:element name="bar" type="xsd:string"/>
        </xsd:sequence>
        <xsd:element name="ding" type="xsd:string"/>
    </xsd:choice>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record E34a {
    union {
        record {
            XSD.String foo,
            XSD.String bar
        } sequence,
        XSD.String ding
    } choice
}
with {
    variant "name as uncapitalized ";
    variant(choice, choice.sequence) "untagged";
}
```

EXAMPLE 2: Multiple *sequence-s* nested to *choice*:

```
<xsd:complexType name="e34b">
    <xsd:choice>
        <xsd:sequence>
            <sequence>
                <xsd:element name="foo" type="xsd:string"/>
                <xsd:element name="bar" type="xsd:string"/>
            </xsd:sequence>
            <xsd:element name="ding" type="xsd:string"/>
            <xsd:element name="foo" type="xsd:string"/>
            <xsd:element name="bar" type="xsd:string"/>
        </xsd:sequence>
        <xsd:element name="ding" type="xsd:string"/>
    </xsd:choice>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record E34b {
    union {
        record {
            XSD.String foo,
            XSD.String bar,
            XSD.String ding,
            XSD.String foo_1,
            XSD.String bar_1
        } sequence,
        XSD.String ding
    } choice
}
with {
    variant "name as uncapitalized ";
    variant(foo_) "name as 'foo'";
    variant(bar_) "name as 'bar'";
    variant(choice, choice.sequence) "untagged";
    variant (choice.sequence.foo_1) "name as 'foo'";
    variant (choice.sequence.bar_1) "name as 'bar'";
}
```

7.6.5.5 Choice with nested any

An XSD *any* element nested to a *choice* shall be translated according to clause 7.7.

EXAMPLE:

```
<xsd:complexType name="e35">
    <xsd:choice>
        <xsd:element name="foo" type="xsd:string"/>
        <xsd:any namespace="other"/>
    </xsd:choice>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record E35 {
    union {
        XSD.String foo,
        XSD.String elem
    } choice
```

```
}
with {
    variant "name as uncapitalized";
    variant(choice) "untagged";
    variant(choice.elem) "anyElement from 'other' ";
}
```

7.6.6 Sequence content

7.6.6.0 General

An XSD *sequence* defines an ordered collection of components and its content may be of any combination of XSD *element*s, *group* references, *choice*, *sequence* or *any*.

Clauses 7.6.6.1 to 7.6.6.5 discuss the mapping for various contents nested in an XSD *sequence* component in the general case, when both the *minOccurs* and *maxOccurs* attribute equal to "1" (either explicitly or by defaulting to "1").

Clause 7.6.6.6 describes the mapping when either the *minOccurs* or the *maxOccurs* attribute of the sequence compositor or both do not equal to "1".

7.6.6.1 Sequence with nested element content

In the general case, child elements of a *sequence*, which is a child of a *complexType*, shall be mapped to TTCN-3 as fields of the enframing **`record`** (see clause 7.6) (i.e. the *sequence* itself is not producing any TTCN-3 construct).

EXAMPLE: Mapping a mandatory *sequence* content:

```
<xsd:complexType name="e36a">
    <xsd:sequence>
        <xsd:element name="foo" type="xsd:integer"/>
        <xsd:element name="bar" type="xsd:float"/>
    </xsd:sequence>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record E36a {
    XSD.Integer foo,
    XSD.Float  bar
}
with {
    variant "name as uncapitalized";
}
```

7.6.6.2 Sequence with nested group content

In the general case, nested group reference components shall be mapped to a field of the enframing **`record`** type (see clause 7.6) or field. The type of the field shall be the TTCN-3 type generated for the referenced group and the name of the field shall be the result of applying clause 5.2.2 to the *name* of the referenced group.

EXAMPLE: The following example shows this translation with a *choice* group and an *element*:

```
<xsd:group name="e37">
    <xsd:choice>
        <xsd:element name="foo" type="xsd:string"/>
        <xsd:element name="bar" type="xsd:string"/>
    </xsd:choice>
</xsd:group>

<xsd:complexType name="e38">
    <xsd:sequence>
        <xsd:group ref="ns:e37"/>
        <xsd:element name="ding" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type union E37 {
    XSD.String foo,
    XSD.String bar
}
with {
    variant "name as uncapitalized";
    variant "untagged";
}

type record E38 {
    E37 e37,
    XSD.String ding
}
with {
    variant "name as uncapitalized";
}
```

7.6.6.3 Sequence with nested choice content

An XSD *choice* nested to a *sequence* shall be mapped as a field of the enframing **record** (see clauses 7.6, 7.6.5.4 and 7.6.6.4), according to clause 7.6.5 (i.e. the *sequence* itself is not producing any TTCN-3 construct).

EXAMPLE:

```
<xsd:complexType name="e39">
    <xsd:sequence>
        <xsd:choice>
            <xsd:element name="foo" type="xsd:string"/>
            <xsd:element name="bar" type="xsd:string"/>
        </xsd:choice>
        <xsd:element name="ding" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record E39 {
    union {
        XSD.String foo,
        XSD.String bar
    } choice,
    XSD.String ding
}
with {
    variant "name as uncapitalized";
    variant(choice) "untagged";
}
```

7.6.6.4 Sequence with nested sequence content

In the general case (i.e. when implicitly or explicitly minOccurs=maxOccurs=1), components of a *sequence* nested in a *sequence* shall be translated to TTCN-3 according to clauses 7.6.6.1 to 7.6.6.3, 7.6.6.5 and 7.6.6.6 and the resulted constructs shall be added to the enframing (outer) **record** type or field (see also clauses 7.6 and 7.6.5.4).

EXAMPLE 1: Sequence nesting a mandatory *sequence*:

```
<xsd:complexType name="e40a">
    <xsd:sequence>
        <xsd:sequence>
            <xsd:element name="foo" type="xsd:string"/>
            <xsd:element name="bar" type="xsd:string"/>
        </xsd:sequence>
        <xsd:element name="ding" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record E40a {
    XSD.String foo,
    XSD.String bar,
```

```
    XSD.String ding
}
with {
    variant "name as uncapitalized";
}
```

EXAMPLE 2: Sequence nesting another sequence, choice and an additional element:

```
<xsd:complexType name="e40b">
    <xsd:sequence>
        <xsd:sequence>
            <xsd:element name="foo" type="xsd:string"/>
            <xsd:element name="bar" type="xsd:string"/>
        </xsd:sequence>
        <xsd:choice>
            <xsd:element name="foo" type="xsd:string"/>
            <xsd:element name="bar" type="xsd:string"/>
        </xsd:choice>
        <xsd:element name="ding" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record E40b {
    XSD.String foo,
    XSD.String bar,
    union {
        XSD.String foo,
        XSD.String bar
    } choice,
    XSD.String ding
}
with {
    variant "name as uncapitalized";
    variant(choice) "untagged";
}
```

7.6.6.5 Sequence with nested any content

An XSD *any* element nested in a *sequence* shall be translated according to clause 7.7.

EXAMPLE:

```
<xsd:complexType name="e41">
    <xsd:sequence>
        <xsd:element name="foo" type="xsd:string"/>
        <xsd:any/>
    </xsd:sequence>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record E41 {
    XSD.String foo,
    XSD.String elem
}
with {
    variant "name as uncapitalized";
    variant(elem) "anyElement";
}
```

7.6.6.6 Effect of the *minOccurs* and *maxOccurs* attributes on the mapping

When either or both the *minOccurs* and/or the *maxOccurs* attributes of the *sequence* compositor specify a different value than "1", the following rules shall apply:

a) First, the *sequence* compositor shall be mapped to a TTCN-3 **`record`** field (as opposed to ignoring it in the previous clauses, when both *minOccurs* and *maxOccurs* equal to 1) with the name "sequence".

b) The encoding instruction "untagged" shall be attached to the field corresponding to *sequence*.

c) The procedures in clause 7.1.4 shall be applied to this **`record`** field.

NOTE: As the result of applying clause 7.1.4, the type of the field may be changed to **`record of record`** and in parallel the name of the field should be changed to "sequence_list".

d) Finally, clause 5.2.2 shall be applied to the name of the resulted field and the field shall be added to the enframing TTCN-3 **record** (see clauses 7.6 and 7.6.6) or **`union`** field (see clause 7.6.5).

EXAMPLE 1: Mapping an optional *sequence*:

```
<xsd:complexType name="e36b">
    <xsd:sequence minOccurs="0">
        <xsd:element name="foo" type="xsd:integer"/>
        <xsd:element name="bar" type="xsd:float"/>
    </xsd:sequence>
</xsd:complexType>
```

Will be mapped to TTCN-3 e.g. as:

```
type record E36b {
    record {
        XSD.Integer foo,
        XSD.Float  bar
    } sequence optional
}
with {
    variant "name as uncapitalized";
    variant (sequence) "untagged";
}
```

EXAMPLE 2: Sequence nesting an optional sequence:

```
<xsd:complexType name="e40c">
    <xsd:sequence>
        <sequence minOccurs="0">
            <xsd:element name="foo" type="xsd:string"/>
            <xsd:element name="bar" type="xsd:string"/>
        </xsd:sequence>
        <xsd:choice>
            <xsd:element name="foo1" type="xsd:string"/>
            <xsd:element name="bar1" type="xsd:string"/>
        </xsd:choice>
        <xsd:element name="ding" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
```

Will be mapped to TTCN-3 e.g. as:

```
type record E40c {
    record {
        XSD.String foo,
        XSD.String bar
    } sequence optional,
    union {
        XSD.String foo1,
        XSD.String bar1
    } choice,
    XSD.String ding
}
with {
    variant "name as uncapitalized";
    variant(sequence, choice) "untagged";
}
```

EXAMPLE 3: Sequence nesting a sequence of multiple recurrence:

```
<xsd:complexType name="e40d">
    <xsd:sequence>
        <xsd:sequence minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="foo" type="xsd:string"/>
            <xsd:element name="bar" type="xsd:string"/>
        </xsd:sequence>
        <xsd:element name="ding" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
```

Will be mapped to TTCN-3 e.g. as:

```
type record E40d {
    record of record {
        XSD.String foo,
        XSD.String bar
    } sequence_list,
    XSD.String ding
}
with {
    variant "name as uncapitalized";
    variant (sequence_list) "untagged";
}
```

EXAMPLE 4: Decoding an empty XML *element* when the optional *sequence* contains optional elements only:

```
<xsd:element name="optionals_in_optional">
    <xsd:complexType>
        <xsd:sequence minOccurs="0">
            <xsd:element name="elem1" type="xsd:string" minOccurs="0"/>
            <xsd:element name="elem2" type="xsd:integer" minOccurs="0"/>
            <xsd:element name="elem3" type="xsd:decimal" minOccurs="0"/>
            <xsd:element name="elem4" type="xsd:dateTime" minOccurs="0"/>
            <xsd:element name="elem5" type="xsd:duration" minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Will be mapped to TTCN-3 e.g. as:

```
type record Optionals_in_optional
{
    record {
        XSD.String elem1 optional,
        XSD.Integer elem2 optional,
        XSD.Decimal elem3 optional,
        XSD.DateTime elem4 optional,
        XSD.Duration elem5 optional
    } sequence optional
}
with {
    variant "name as uncapitalized";
    variant "element";
    variant (sequence) "untagged";
};
```

And an incoming empty element, e.g. <optionals_in_optional></optionals_in_optional> will be decoded to the short TTCN-3 value (see clause B.3.21):

```
{ Optionals_in_optional := { sequence := omit }};
```

7.6.7 Attribute definitions, attribute and attributeGroup references

Locally defined *attribute* elements, references to global *attribute* elements and references to *attributeGroup*s shall be mapped jointly. XSD attributes, either local or referenced global (including the **content** of referenced attributeGroups) shall be mapped to individual fields of the enframing TTCN-3 **`record`** (see clause 7.6) directly (i.e. without nesting). The types of the fields shall be the types of the corresponding attributes, mapped to TTCN-3 the same way as specified in clause 7.4.1 for global *attribute* elements, and the names of the fields shall be the names resulted in applying clause 5.2.2 to the attribute names. The fields generated for local attribute definitions, references and contents of referenced attribute groups shall be inserted in the following order: they shall first be ordered, in an ascending alphabetical order, by the target namespaces of the attribute declarations, with the fields without a target namespace preceding fields with a target namespace, and then by the names of the attribute declarations within each target namespace (also in ascending alphabetical order).

XSD local attribute declarations and references may contain also the special attribute *use*. The above mapping shall be carried out jointly with the procedures specified for the *use* attribute in clause 7.1.12.

TTCN-3 **`record`** fields generated for *attribute* element or *attributeGroup* references, where the namespace of the referenced XSD entity differs from the target namespace of the referencing XSD schema (including the no target namespace case), shall be appended with a "namespace as" encoding instruction (see clause B.3.1), which shall identify the namespace and optionally the prefix of the XSD schema in which the referenced entity is defined.

All generated TTCN-3 fields shall also be appended with the "attribute" encoding instruction.

EXAMPLE 1: Referencing an *attributeGroup* in a *complexType*:

```
<xsd:attributeGroup name="e42">
    <xsd:attribute name="foo" type="xsd:float"/>
    <xsd:attribute name="bar" type="xsd:float"/>
</xsd:attributeGroup>

<xsd:complexType name="e44">
    <xsd:sequence>
        <xsd:element name="ding" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attributeGroup ref="ns:e42"/>
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type record E44 {
    XSD.Float bar optional,
    XSD.Float foo optional,
    XSD.String ding
}
with {
    variant "name as uncapitalized";
    variant (bar,foo) "attribute";
}
```

EXAMPLE 2: Mapping of a local attributes, attribute references and attribute group references without a target namespace:

```
<xsd:attribute name="fooGlobal" type="xsd:float" />
<xsd:attribute name="barGlobal" type="xsd:string" />
<xsd:attribute name="dingGlobal" type="xsd:integer" />

<xsd:attributeGroup name="Agroup">
    <xsd:attribute name="fooInAgroup" type="xsd:float" />
    <xsd:attribute name="barInAgroup" type="xsd:string" />
    <xsd:attribute name="dingInAgroup" type="xsd:integer" />
</xsd:attributeGroup>

<xsd:complexType name="e17A">
    <xsd:sequence>
        <xsd:element name="elem" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute ref="fooGlobal" />
    <xsd:attribute ref="barGlobal" />
    <xsd:attribute ref="dingGlobal" />
    <xsd:attribute name="fooLocal" type="xsd:float" />
    <xsd:attribute name="barLocal" type="xsd:string" />
    <xsd:attribute name="dingLocal" type="xsd:integer" />
    <xsd:attributeGroup ref="Agroup" />
</xsd:complexType>
```

Will be translated to TTCN-3 e.g. as:

```
type XSD.Float FooGlobal
with {
    variant "name as uncapitalized ";
    variant "attribute";
}

type XSD.String BarGlobal
with {
    variant "name as uncapitalized ";
    variant "attribute";
}

type XSD.Integer DingGlobal
with {
    variant "name as uncapitalized ";
    variant "attribute";
}

type record E17A {
    XSD.String barGlobal optional,
    XSD.String barInAgroup optional,
```

```
        XSD.String barLocal optional,
        XSD.Integer dingGlobal optional,
        XSD.Integer dingInAgroup optional,
        XSD.Integer dingLocal optional,
        XSD.Float fooGlobal optional,
        XSD.Float fooInAgroup optional,
        XSD.Float fooLocal optional,
        XSD.String elem
    }
    with {
        variant "name as uncapitalized ";
        variant(barGlobal,barInAgroup,barLocal,dingGlobal,dingInAgroup,dingLocal,fooGlobal,
                fooInAgroup,fooLocal) "attribute";
    }
```

EXAMPLE 3: Mapping the same local attributes, attribute references and attribute group references as above but with a target schema namespace:

Using the same global attribute, attribute group and complex type definitions as in the previous example, e17A is translated to TTCN-3 as:

```
    type record E17A {
        XSD.Float barInAgroup optional,
        XSD.String barLocal optional,
        XSD.Integer dingInAgroup optional,
        XSD.Integer dingLocal optional,
        XSD.Float fooInAgroup optional,
        XSD.Float fooLocal optional,
        XSD.String barGlobal optional,
        XSD.Integer dingGlobal optional,
        XSD.Float fooGlobal optional,
        XSD.String elem
    }
    with {
        variant "name as uncapitalized ";
        variant(barInAgroup,barLocal,dingInAgroup,dingLocal,fooInAgroup,fooLocal,barGlobal,
              dingGlobal,fooGlobal) "attribute";
}
```

7.6.8 Mixed content

When mixed content is allowed for a complex type or content, (i.e. the mixed attribute is set to "true") an additional **`record of XSD.String`** field, with the field name "embed_values" shall be generated and inserted as the first field of the outer enframing TTCN-3 **`record`** type generated for the all, choice or sequence content (see clauses 7.6, 7.6.4, 7.6.5 and 7.6.6). In TTCN-3 values, elements of the `embed_values` field shall be used to provide the actual strings to be inserted into the encoded XML value or extracted from it (the relation between the record of elements and the strings in the encoded XML values is defined in clause B.3.10). In TTCN-3 values the number of components of the `embed_values` field (the number of strings to be inserted) shall not exceed the total number of components present in the enclosing enframing **`record`**, corresponding to the child *element* elements of the complexType with the *mixed*="true" attribute, i.e. ignoring fields corresponding to *attribute* elements, the `embed_values` field itself and the `order` field, if present (see clause 7.6.4), plus 1 (i.e. all components of enclosed **`record of`**-s).

The `embed_values` field shall precede all other fields, resulted by the translation of the *attribute*s and attribute and *attributeGroup* references of the given *complexType* and the `order` field, if any, generated for the *all* content models (see also clause 7.6.4).

EXAMPLE 1: Complex type definition with *sequence* constructor and *mixed* content type:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:ns="http://www.example.org/mixed"
            targetNamespace="http://www.example.org/mixed">
    <xsd:element name="MySeqMixed">
        <xsd:complexType mixed="true">
            <xsd:sequence>
                <xsd:element name="a" type="xsd:string"/>
                <xsd:element name="b" type="xsd:boolean"/>
            </xsd:sequence>
            <xsd:attribute name="attrib" type="xsd:integer"/>
        </xsd:complexType>
    </xsd:element>
```

```
</xsd:schema>
```

Will be translated to the TTCN-3 type definition e.g. as (note that in a TTCN-3 value notation the embed_values field should have max. 3 record of components):

```
module http_www_example_org_mixed {

    import from XSD all;

    type record MySeqMixed {
        record of XSD.String    embed_values,
        XSD.Integer attrib optional,
        XSD.String a,
        XSD.Boolean b
    }
    with {
        variant "element";
        variant "embedValues";
        variant(attrib) "attribute";
    }
}
with {
    encode "XML";
    variant "namespace as 'http://www.example.org/mixed' prefix 'ns'";
    variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

And the template:

```
template MySeqMixed t_MySeqMixed := {
    embed_values:= {"The ordered ", " has arrived ", "Wait for further information."},
    attrib := omit,
    a:= "car",
    b:= true
}
```

Will be encoded in XML, for example, as:

```
<ns:MySeqMixed xmlns:ns='http://www.example.org/mixed'>The ordered <a>car</a> has arrived
<b>true</b>Wait for further information.</ns:MySeqMixed>
```

EXAMPLE 2: Complex type definition with *sequence* constructor of multiple occurrences and *mixed* content type:

```
<xsd:element name="MyComplexElem-16">
    <xsd:complexType mixed="true">
        <xsd:sequence maxOccurs="unbounded" minOccurs="0">
            <xsd:element name="a" type="xsd:string"/>
            <xsd:element name="b" type="xsd:boolean"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Will be translated to the TTCN-3 type definition e.g. as:

```
type record MyComplexTypeElem_16 {
    record of XSD.String    embed_values,
    record of record {
        XSD.String a,
        XSD.Boolean b
    } sequence_list
}
with {
    variant "name as 'MyComplexElem-16'";
    variant "element";
    variant "embedValues";
}
```

And the template:

```
template MyComplexTypeElem_16 t_MyComplexTypeElem_16 := {
    embed_values := {   "The ordered", "has arrived",
                        "the ordered", "Has arrived!", "Wait for further information."},
    sequence_list := {
        { a:= "car", b:= false},
        { a:= "bicycle", b:= true}
```

```
    }
}
```

Will be encoded in XML, for example, as:

```
<ns:MyComplexTypeElem-16>The ordered<a>car</a>has arrived<b>false</b>the ordered
<a>bicycle</a>Has arrived!<b>true</b>Wait for further information.</ns:MyComplexTypeElem-16>
```

EXAMPLE 3: Complex type definition with *all* constructor and *mixed* content type:

```
<xsd:element name="MyComplexElem-13">
    <xsd:complexType mixed="true">
        <xsd:all>
            <xsd:element name="a" type="xsd:string"/>
            <xsd:element name="b" type="xsd:boolean"/>
        </xsd:all>
    </xsd:complexType>
</xsd:element>
```

Will be translated to the TTCN-3 type definition, e.g. as:

```
type record MyComplexTypeElem_13 {
    record of XSD.String embed_values,
    record of enumerated {a,b} order,
    XSD.String a,
    XSD.Boolean b
}
with {
    variant "name as 'MyComplexElem-13'";
    variant "element";
    variant "embedValues";
    variant "useOrder";
}
```

And the template:

```
template MyComplexTypeElem_13 t_MyComplexTypeElem_13 := {
    embed_values:= {"Arrival status", "product name","Wait for further information."},
    order := {b,a},
    a:= "car",
    b:= false
}
```

Will be encoded in XML, for example, as:

```
<ns:MyComplexTypeElem-13>Arrival status<b>false</b>product name<a>car</a>
Wait for further information.</ns:MyComplexTypeElem-13>
```

EXAMPLE 4: Complex type definition with *all* constructor, optional elements and *mixed* content type:

```
<xsd:element name="MyElementMixedOptAll">
    <xsd:complexType mixed="true">
        <xsd:all minOccurs="0">
            <xsd:element name="a" type="xsd:string"/>
            <xsd:element name="b" type="xsd:boolean"/>
        </xsd:all>
    </xsd:complexType>
</xsd:element>
```

Will betranslated to the TTCN-3 type definition e.g. as:

```
type record  MyElementMixedOptAll {
    record of XSD.String embed_values,
    record of enumerated {a,b} order,
    XSD.String a optional,
    XSD.Boolean b optional
}
with {
    variant "element";
    variant "embedValues";
    variant "useOrder";
}
```

And the template:

```
template MyElementMixedOptAll t_MyTemplate := {
    embed_values:= {"Arrival status", "Wait for further information."},
    order := {},
    a:= omit,
    b:= omit
}
```

Will be encoded in XML, for example, as (supposing the target namespace's prefix is "ns"):

```
    <ns:MyElementMixedOptAll>Arrival status</ns:MyElementMixedOptAll>
```

EXAMPLE 5: Complex type definition with *choice* constructor and *mixed* content type:

```
<xsd:element name="MyComplexElem-14">
    <xsd:complexType mixed="true">
        <xsd:choice>
            <xsd:element name="a" type="xsd:string"/>
            <xsd:element name="b" type="xsd:boolean"/>
        </xsd:choice>
    </xsd:complexType>
</xsd:element>
```

Will be translated to the TTCN-3 type definition e.g. as:

```
type record MyComplexTypeElem_14 {
    record of XSD.String embed_values,
    union {
        XSD.String              a,
        XSD.Boolean             b
    } choice
}
with {
    variant "name as 'MyComplexElem-14'";
    variant "element";
    variant "embedValues";
}
```

And the template:

```
template MyComplexTypeElem_14 t_MyComplexTypeElem_14 := {
    embed_values:= {"Arrival status", "Wait for further information."},
    choice := { b:= false }
}
```

Will be encoded in XML, for example, as:

```
<ns:MyComplexTypeElem-14>Arrival status<b>false</b>Wait for further information.
</ns:MyComplexTypeElem-14>
```

7.7 Any and anyAttribute

7.7.0 General

An XSD *any* element can be defined in complex types, as a child of *sequence* or *choice* (i.e. locally only) and specifies that any well-formed XML is permitted in the type's content model. In addition to the *any* element, which enables element content according to namespaces, there is an analogous XSD *anyAttribute* element which enables transparent (from the codec's point of view) attributes to appear in elements.

7.7.1 The any element

The XSD *any* element shall be translated, like other elements, to a field of the enframing **`record`** type or field or **`union`** field (see clauses 7.6, 7.6.5 and 7.6.6). The type of this field shall be `XSD.String` or **`anytype`** (dependent on some tool configuration beyond the scope of the present document) and the name of the field shall be the result of applying clause 5.2.2 to "elem". Finally the "anyElement…" encoding instruction shall be attached, which shall also specify the namespace wildcards and/or list of namespaces which are allowed or restricted to qualify the given element, in accordance with the *namespace* attribute of the XSD *any* element, if present (see details in clause B.3.2).

If the *any* element is mapped to a field of type **anytype**, values or templates of that field shall contain only the type variants XSD.String or the TTCN-3 types translated from XSD *element* definitions allowed by the namespace restriction of the *any* element.

In the translation of *any* XSD elements, when a *processContents* XSD attribute is present, also clause 7.1.15 shall be considered.

NOTE: The mapping may also be influenced by other attributes applied to the component, if any. See more details in clause 7.1, especially clause 7.1.4.

In the value notation the `XSD.String` shall specify a syntactically correct XML element. It shall use a namespace (including the no namespace case) allowed by the final "anyElement" encoding instruction.

EXAMPLE 1: Translating *any* to XSD.String:

The Schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         xmlns:this="http://www.example.org/wildcards"
         targetNamespace="http://www.example.org/wildcards">

<import namespace="http://www.example.org/other" schemaLocation="any_additionalElements.xsd"/>

    <xsd:element name="anyElementOtherNamespace" type="this:e46a"></xsd:element>

    <xsd:element name="e46">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:any namespace="##any"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="e46a">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:any minOccurs="0" namespace="##other"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="e46b">
        <xsd:complexType">
            <xsd:sequence>
                <xsd:any minOccurs="0" maxOccurs="unbounded" namespace="##local"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

</schema>
```

Will be mapped to e.g. the following TTCN-3 module:

```
module http_www_example_org_wildcards {

    import from XSD all;

    type E46a AnyElementOtherNamespace
    with {
        variant "name as uncapitalized";
        variant "element";
    }

    type record E46 {
        XSD.String  elem
    }
    with {
        variant "name as uncapitalized";
        variant "element";
```

```
            variant(elem) "anyElement";
        }

        type record E46a {
            XSD.String  elem optional
        }
        with {
            variant "name as uncapitalized";
            variant "element";
            variant(elem) "anyElement except unqualified,'http://www.example.org/wildcards'";
        }

        type record E46b {
            record of XSD.String elem_list
        }
        with {
            variant "name as uncapitalized";
            variant "element";
            variant(elem_list) "untagged";
            variant (elem_list[-]) "anyElement except unqualified";
        }
    }
    with {
        encode "XML";
        variant "namespace as 'http://www.example.org/wildcards' prefix 'this'";
        variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
    }
```

And the template:

```
    module EncDec_checking {

    import from http_www_example_org_wildcards all;

    template AnyElementOtherNamespace t_AnyElementOtherNamespace := {
      elem := "<other:valami xmlns:other=""http://www.example.org/other"">text</other:valami>"
    }

    }//end module
```

Note the following subsidiary schema, used by the encoded XML value:

```
    <?xml version="1.0" encoding="UTF-8"?>
    <xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
            targetNamespace="http://www.example.org/other">
        <xsd:element name="valami" type="xsd:string"/>
    </xsd:schema>
```

Should be encoded e.g. to the following XML instance:

```
    <?xml version="1.0" encoding="UTF-8"?>
    <this:anyElementOtherNamespace xmlns:this='http://www.example.org/wildcards'>
    <other:valami xmlns:other="http://www.example.org/other">text</other:valami>
    </this:anyElementOtherNamespace>
```

While, for example, receiving the following XML instance is causing a decoding failure, because the XML element used in place of the any element shall be from a namespace different from "http://www.example.org/wildcards":

```
    <?xml version="1.0" encoding="UTF-8"?>
    <this:anyElementOtherNamespace xmlns:this='http://www.example.org/wildcards'>
    <other:valami xmlns:other="http://www.example.org/wildcards">text</other:valami>
    </this:anyElementOtherNamespace>
```

EXAMPLE 2: Translating *any* to **anytype**:

The Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         xmlns:this="http://www.example.org/wildcards"
         targetNamespace="http://www.example.org/wildcards">

<import namespace="http://www.example.org/other" schemaLocation="any_additionalElements.xsd"/>

<xsd:element name="anyElementOtherNamespace" type="this:e46a"></xsd:element>
```

```
<xsd:element name="e46">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:any namespace="##any"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="e46a">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:any minOccurs="0" namespace="##other"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="e46b">
    <xsd:complexType">
        <xsd:sequence>
            <xsd:any minOccurs="0" maxOccurs="unbounded" namespace="##local"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

</schema>
```

Will be mapped e.g. to the following TTCN-3 module:

```
module http_www_example_org_wildcards {

    import from XSD all;

    type E46a AnyElementOtherNamespace
    with {
        variant "name as uncapitalized";
        variant "element"
    }

    type record E46 {
        anytype  elem
    }
    with {
        variant "name as uncapitalized";
        variant "element";
        variant(elem) "anyElement"
    }

    type record E46a {
        anytype  elem optional
    }
    with {
        variant "name as uncapitalized";
        variant "element";
        variant(elem) "anyElement except unqualified,'http://www.example.org/wildcards'"
    }

    type record E46b {
        record of anytype elem_list
    }
    with {
        variant "name as uncapitalized";
        variant "element";
        variant(elem_list) "untagged"
        variant (elem_list[-]) "anyElement except unqualified"
    }
}
with {
    encode "XML";
    variant "namespace as 'http://www.example.org/wildcards' prefix 'this'";
    variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

And the template:

```
module EncDec_checking {

import from http_www_example_org_wildcards all;
import from http_www_example_org_other all;
```

```
template AnyElementOtherNamespace t_AnyElementOtherNamespace := {
  elem := {Valami := "text"}
}

}//end module

Note the following subsidiary schema, used by the encoded XML value:
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
        targetNamespace="http://www.example.org/other">
    <xsd:element name="valami" type="xsd:string"/>
</xsd:schema>
```

Should be encoded e.g. to the following XML instance:

```
<?xml version="1.0" encoding="UTF-8"?>
<this:anyElementOtherNamespace xmlns:this='http://www.example.org/wildcards'>
<other:valami xmlns:other="http://www.example.org/other">text</other:valami>
</this:anyElementOtherNamespace>
```

While, for example, receiving the following XML instance is causing a decoding failure, because the XML element used in place of the any element shall be from a namespace different from "http://www.example.org/wildcards":

```
<?xml version="1.0" encoding="UTF-8"?>
<this:anyElementOtherNamespace xmlns:this='http://www.example.org/wildcards'>
<other:valami xmlns:other="http://www.example.org/wildcards">text</other:valami>
</this:anyElementOtherNamespace>
```

7.7.2 The anyAttribute element

The *anyAttribute* element shall be translated, like other attributes, to a field of the enframing **`record`** type or field or **`union`** field (see clauses 7.6, 7.6.5 and 7.6.6). The type of this field shall be **`record length`** `(1..`**`infinity`**`)` **`of`** `XSD.String` or **`record length (1..infinity) of anytype`** (dependent on some tool configuration beyond the scope of the present document), the field shall always be **`optional`** and the name of the field shall be the result of applying clause 5.2.2 to "attr". In the case an XSD component contains more than one *anyAttribute* elements (e.g. by a complex type extending an another complex type already containing an *anyAttribute*), only one new field shall be generated for all the *anyAttribute* elements (with the name resulted from applying clause 5.2.2 to "attr") but the namespace specifications of all *anyAttribute* components shall be considered in the "anyAttributes" encoding instruction (see below). The field shall be inserted directly after the fields generated for the XSD *attribute* elements of the same component or, if the component does not contain an *attribute* component, in the place where the first field generated for an XSD *attribute* would be inserted (see clause 7.6.7).

If the *anyAttribute* element is translated to a field of type **`record length (1..infinity) of anytype,`** the field shall contain only the type variants XSD.String or types that have been translated from those XSD *attribute* definitions allowed by the namespace restriction of the *anyAttribute* element.

Finally the " anyAttributes …" encoding instruction (see clause B.3.3) shall be attached, which shall also specify the namespace wildcards and/or list of namespaces which are allowed or restricted to qualify the given element, in accordance with the *namespace* attribute of the XSD *anyAttribute* element if present (see details in clause B.3.3).

NOTE 1: When translating XSD *attribute* elements, the *use* attribute determines if the generated field is **`optional`** or not (see clause 7.1.12). Because the *use* attribute is not allowed for *anyAttribute* elements, the generated record of field will always be optional.

In the translation of *anyAttribute* XSD elements, when a *processContents* XSD attribute is present, also clause 7.1.15 shall be considered.

In the value notation, when mapping to record of XSD.String or when using anytype type variant XSD.String each `XSD.String` of the generated **`record of`** shall specify exactly one XML attribute using the following format: it shall be composed of an optional URI followed by whitespace, followed by the non-qualified name of the XML attribute, followed by an EQUALS SIGN (=) character, followed by a APOSTROPHE (') character or two QUOTATION MARK (") characters, followed by the XML attribute value, followed by a APOSTROPHE (') character or two QUOTATION MARK (") characters. In the string there shall be no other whitespace than specified above. Each string shall use a namespace (including the no namespace case) allowed by the final "anyAttributes" encoding instruction.

NOTE 2: The metaformat of each `XSD.String` is:
"[<URI><whitespace>]<non-qualified attribute name>=('|"")< attribute value>('|"")".

NOTE 3: Decoders are always using a single SPACE character as whitespace between the URI and the non-qualified attribute name parts of the string (see clause B.3.3) to allow the user to employ specific values for matching.

EXAMPLE 1: Translating *anyAttribute* to XSD.String:

The Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         xmlns:this="http://www.example.org/wildcards"
         targetNamespace="http://www.example.org/wildcards">

    <xsd:element name="anyAttrAnyNamespace" type="this:e45"/>

    <xsd:element name="anyAttrThisNamespace" type="this:e45b"/>
        <xsd:complexType name="e45">
            <xsd:attribute name="attr" type="xsd:string"/>
            <xsd:attribute name="bb" type="xsd:date"/>
            <xsd:attribute name="aa" type="xsd:date"/>
            <xsd:anyAttribute namespace="##any"/>
        </xsd:complexType>      </xsd:element>

    <xsd:element name="e45a">
        <xsd:complexType>
            <xsd:anyAttribute namespace="##other"/>
        </xsd:complexType>
</xsd:element>

    <xsd:element name="e45b">
        <xsd:complexType>
            <xsd:anyAttribute namespace="##targetNamespace"/>
        </xsd:complexType>
   </xsd:element>

    <xsd:element name="e45c">
        <xsd:complexType>
            <xsd:anyAttribute namespace="##local http://www.example.org/attribute"/>
        </xsd:complexType>
   </xsd:element>

    <xsd:element name="e45d">
        <xsd:complexType>
            <xsd:complexContent>
                <xsd:extension base="this:e45c">
                    <xsd:anyAttribute namespace="##targetNamespace"/>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
   </xsd:element>

</xsd:schema>
```

Will be mapped e.g. to the following TTCN-3 module:

```
module http_www_example_org_wildcards {

    import from XSD all;

    type E45 AnyAttrAnyNamespace
    with {
        variant "name as uncapitalized";
        variant "element";
    }

    type E45b AnyAttrThisNamespace
    with {
        variant "name as uncapitalized";
        variant "element";
    }

    type record E45 {
        XSD.Date   aa optional,
```

```
        XSD.String attr optional,
        XSD.Date   bb optional,
        record length (1..infinity) of XSD.String attr_1 optional
    }
    with {
        variant "name as uncapitalized";
        variant "element";
        variant (aa, attr, bb) "attribute";
        variant (attr_1) "anyAttributes";
    }

    type record E45a {
        record length (1..infinity) of XSD.String attr optional
    }
    with {
        variant "name as uncapitalized";
        variant "element";
        variant (attr) "anyAttributes except    unqualified,
                        'http://www.example.org/wildcards'";
    }

    type record E45b {
        record length (1..infinity) of XSD.String attr optional
    }
    with {
        variant "name as uncapitalized";
        variant "element";
        variant(attr) "anyAttributes from ' http://www.example.org/wildcards'";
    }

    type record E45c {
        record length (1..infinity) of XSD.String attr optional
    }
    with {
        variant "name as uncapitalized";
        variant "element";
        variant (attr) "anyAttributes from unqualified,' http://www.example.org/wildcards'";
    }

    type record E45d {
        record length (1..infinity) of XSD.String attr optional
    }
    with {
        variant "name as uncapitalized";
        variant "element";
        variant (attr) "anyAttributes from unqualified, ' http://www.example.org/wildcards',
                       ' http://www.example.org/wildcards'";
    }
} //end module
with {
    encode "XML";
    variant "namespace as 'http://www.example.org/wildcards' prefix 'this'";
    variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

For example the template:

```
    template AnyAttrThisNamespace t_AnyAttrThisNamespace := {
      attr := omit
    }
```

Should be encoded as an empty element with no attribute in XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<this:anyAttrThisNamespace xmlns:this='http://www.example.org/wildcards'/>
```

And the template:

```
    template AnyAttrThisNamespace t_AnyAttrThisNamespace := {
  attr := {"http://www.example.org/wildcards akarmi='tinky-winky'",
           "http://www.example.org/wildcards valami='dipsy'"}
}
```

Should be encoded e.g. to one of the following XML instances:

```
 <?xml version="1.0" encoding="UTF-8"?>
 <this:anyAttrThisNamespace xmlns:this='http://www.example.org/wildcards'
xmlns:b0='http://www.example.org/wildcards' b0:akarmi='tinky-winky'
xmlns:b1='http://www.example.org/wildcards' b0:valami='dipsy'/>
```

Or

```
     <?xml version="1.0" encoding="UTF-8"?>
 <this:anyAttrThisNamespace xmlns:this="http://www.example.org/wildcards"
  this:akarmi="tinky-winky" this:valami="dipsy"/>
```

While, for example, receiving the following XML instance shall cause a decoding failure, because all XML attributes shall be from the namespace "http://www.example.org/wildcards":

```
 <?xml version="1.0" encoding="UTF-8"?>
 <this:anyAttrThisNamespace xmlns:this="http://www.example.org/wildcards"
  xmlns:other="http://www.example.org/other "
  this:akarmi="tinky-winky" other:valami="dipsy"/>
```

EXAMPLE 2: Translating *anyAttribute* to **`anytype`**:

The Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         xmlns:this="http://www.example.org/wildcards"
         targetNamespace="http://www.example.org/wildcards">

<xsd:element name="anyAttrAnyNamespace" type="this:e45"/>

<xsd:element name="anyAttrThisNamespace" type="this:e45b"/>
    <xsd:complexType name="e45">
        <xsd:attribute name="attr" type="xsd:string"/>
        <xsd:attribute name="bb" type="xsd:date"/>
        <xsd:attribute name="aa" type="xsd:date"/>
        <xsd:anyAttribute namespace="##any"/>
    </xsd:complexType>

    <xsd:element name="e45a">
        <xsd:complexType>
            <xsd:anyAttribute namespace="##other"/>
       </xsd:complexType>   </xsd:element>


    <xsd:element name="e45b">
        <xsd:complexType>
            <xsd:anyAttribute namespace="##targetNamespace"/>
       </xsd:complexType>
     </xsd:element>

    <xsd:element name="e45c">
        <xsd:complexType>
            <xsd:anyAttribute namespace="##local http://www.example.org/attribute"/>
       </xsd:complexType>
     </xsd:element>

    <xsd:element name="e45d">
        <xsd:complexType>
            <xsd:complexContent>
                <xsd:extension base="ns:e45c">
                    <xsd:anyAttribute namespace="##targetNamespace"/>
                </xsd:extension>
            </xsd:complexContent>
       </xsd:complexType>
     </xsd:element>

</xsd:schema>
```

Will be mapped e.g. to the following TTCN-3 module:

```
module http_www_example_org_wildcards {

    import from XSD all;

    type E45 AnyAttrAnyNamespace
    with {
        variant "name as uncapitalized";
        variant "element";
    }


    type E45b AnyAttrThisNamespace
    with {
        variant "name as uncapitalized";
        variant "element";
    }

    type record E45 {
        XSD.Date   aa optional,
        XSD.String attr optional,
        XSD.Date   bb optional
        record length (1..infinity) of anytype attr_1 optional
   }
    with {
        variant "name as uncapitalized";
        variant "element";
        variant(aa, attr, bb) "attribute";
        variant(attr_1) "anyAttributes"
   }

   type record E45a {
        record length (1..infinity) of anytype attr optional
   }
    with {
        variant "name as uncapitalized";
        variant "element";
        variant(attr) "anyAttributes except unqualified,'http://www.example.org/wildcards'"
   }

   type record E45b {
        record length (1..infinity) of anytype attr optional
   }
    with {
        variant "name as uncapitalized";
        variant "element";
        variant(attr) "anyAttributes from ' http://www.example.org/wildcards'"
   }

    type record E45c {
        record length (1..infinity) of anytype attr optional
    }
    with {
        variant "name as uncapitalized";
        variant "element";
        variant(attr) "anyAttributes from unqualified,' http://www.example.org/wildcards'"
   }


   type record E45d {
        record length (1..infinity) of anytype attr optional
   }
    with {
        variant "name as uncapitalized";
        variant "element";
        variant(attr) "anyAttributes from unqualified, ' http://www.example.org/wildcards',
                      ' http://www.example.org/wildcards'"
    }
} //end module
with {
    encode "XML";
    variant "namespace as 'http://www.example.org/wildcards' prefix 'this'";
    variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

For example the template:

```
template AnyAttrThisNamespace t_AnyAttrThisNamespace := {
  attr := omit
}
```

Shall be encoded as an empty element with no attribute in XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<this:anyAttrThisNamespace xmlns:this='http://www.example.org/wildcards'/>
```

And the template:

```
template AnyAttrThisNamespace t_AnyAttrThisNamespace := {
  attr := {{String:=http://www.example.org/wildcards akarmi='tinky-winky'},
           {Valami := "dipsy"}}
}
```

Should be encoded e.g. to one of the following XML instances:

```
<?xml version="1.0" encoding="UTF-8"?>
<this:anyAttrThisNamespace xmlns:this='http://www.example.org/wildcards'
xmlns:b0='http://www.example.org/wildcards' b0:akarmi='tinky-winky'
xmlns:b1='http://www.example.org/wildcards' b0:valami='dipsy'/>
```

Or

```
<?xml version="1.0" encoding="UTF-8"?>
<this:anyAttrThisNamespace xmlns:this="http://www.example.org/wildcards"
 this:akarmi="tinky-winky" this:valami="dipsy"/>
```

While, for example, receiving the following XML instance shall cause a decoding failure, because all XML attributes shall be from the namespace "http://www.example.org/wildcards":

```
<?xml version="1.0" encoding="UTF-8"?>
<this:anyAttrThisNamespace xmlns:this="http://www.example.org/wildcards"
 xmlns:other="http://www.example.org/other "
 this:akarmi="tinky-winky" other:valami="dipsy"/>
```

7.8 Annotation

An XSD *annotation* is used to include additional information in the XSD data. Annotations may appear in every component and shall be mapped to a corresponding comment in TTCN-3. The comment shall appear in the TTCN-3 code just before the mapped structure it belongs to. The present document does not describe a format in which the comment shall be inserted into the TTCN-3 code.

EXAMPLE:

```
<xsd:annotation>
    <xsd:appinfo>Note</xsd:appinfo>
    <xsd:documentation xml:lang="en">This is a helping note!</xsd:documentation>
</xsd:annotation>
```

Could be translated to:

```
// Note: This is a helping note !
```

7.9 Group components

XSD *group* definition, defined globally, enables groups of elements to be defined and named, so that the elements can be used to build up the content models of complex types. The child of a group shall be one of the *all*, *choice* or *sequence* compositors.

They shall be mapped to TTCN-3 type definitions the same way as their child components would be mapped inside a *complexType* with one difference: the "untagged" encoding instruction shall be attached to the generated TTCN-3 component, corresponding to the *group* element.

EXAMPLE: Mapping of groups:

```
<xsd:group name="shipAndBill">
  <xsd:sequence>
    <xsd:element name="shipTo" type="xsd:string"/>
    <xsd:element name="billTo" type="xsd:string"/>
  </xsd:sequence>
</xsd:group>

<xsd:group name="shipOrBill">
  <xsd:choice>
    <xsd:element name="shipTo" type="xsd:string"/>
    <xsd:element name="billTo" type="xsd:string"/>
  </xsd:choice>
</xsd:group>

<xsd:group name="shipAndBillAll">
  <xsd:all>
    <xsd:element name="shipTo" type="xsd:string"/>
    <xsd:element name="billTo" type="xsd:string"/>
  </xsd:all>
</xsd:group>
```

Will be translated to TTCN-3 e.g. as:

```
type record ShipAndBill {
    XSD.String shipTo,
    XSD.String billTo
}
with {
    variant "untagged";
}

type union ShipOrBill {
    XSD.String shipTo,
    XSD.String billTo
}
with {
    variant "untagged";
}

type record ShipAndBillAll {
    record of enumerated { shipTo, billTo } order,
    XSD.String shipTo,
    XSD.String billTo
}
with {
    variant "untagged";
    variant "useOrder";
}
```

7.10 Identity-constraint definition schema components

The XSD *unique* element enables to indicate that some XSD attribute or element values shall be unique within a certain scope. As TTCN-3 does not allow a similar relational value constraint, mapping of the *unique*, *key* and *keyref* elements are not supported by the present document, i.e. these elements shall be ignored in the translation process.

NOTE 1: It is recommended that converter tools retain the information of the *unique*, *key* and *keyref* elements in a TTCN-3 comment, to help the user in producing TTCN-3 values and templates complying to the original XSD specification.

NOTE 2: As the *selector* and *field* XSD elements may only appear as child elements of a *unique*, *key* or *keyref* element, they are automatically ignored when their parent element is ignored.

8 Substitutions

8.0 General

XSD allows two types of substitutions:

- XML elements in instance documents may be replaced by other XML elements that have been declared as members of the substitution group in XSD (of which the replaced *element* is the head); both the head element and the substitution group members shall be global XSD *elements*; the types of the substitution group members shall be the same or derived from the type of the head element.
- The XSD type actually used to create the instance of an XSD *element* information item may also be a named simple or complex type derived from the type referenced by the *type* attribute of the XSD *element* information item declaration; in this case the *xsi:type* (schema instance namespace) XML attribute shall identify the name of the type used to create the given instance.

Depending on the SUT to be tested, it may be known a priori if the SUT could use element and/or type substitution or not. For this reason, to simplify the generated TTCN-3 code in certain cases, TTCN-3 tools claiming to conform with the present document shall support the following modes of operation, selectable by the user:

- generate a TTCN-3 code allowing both element substitution (code generated according to clause 8.1) and allowing type substitution (code generated according to clause 8.2);
- generate a TTCN-3 code allowing element substitution (code generated according to clause 8.1) but disallowing type substitution (code generated according to clauses 7.5 and 7.6);
- generate a TTCN-3 code disallowing element substitution (code generated according to clauses 7.3 and 8.1.2) but allowing type substitution (code generated according to clause 8.2);
- generate a TTCN-3 code disallowing both element and type substitutions; for backward compatibility with the previous versions of the present document this shall be the default mode.

8.1 Element substitution

8.1.1 Head elements of substitution groups

This clause is invoked if the global XSD *element* information item being translated is referenced by the *substitutionGroup* attribute of one or more other global *element* information item(s) in the set of schemas being translated (i.e. it is the head of an element substitution group) and the user has requested to generate TTCN-3 code allowing using element substitution (see clause 8).

Substitution group head elements shall be translated to TTCN-3 **`union`** types. The name of the **`union`** type shall be the result of applying clause 5.2.2 to the name composed of the header element's name and the postfix "_group".

One alternative shall be added for the head element itself and one alternative for each member of the substitution group. The first alternative (field) of the **`union`** type shall be a default alternative (see clause 6.2.5 of ETSI ES 201 873-1 [1]), and it shall correspond to the head element. The alternatives corresponding to the member elements shall be added in an ordered manner, first alphabetically ordering the elements according to their target namespaces (elements with no target namespace first) and subsequently alphabetically ordering the elements with the same namespace based on their names. For each alternative the field name shall be the name applying clause 5.2.2 to the name of the XSD *element* corresponding to the given alternative. The type of the alternative shall be:

- the TTCN-3 type resulted by applying clause 7.3 to the head element, in the case of the head element;
- the TTCN-3 type resulted by applying clause 8.1.2 to the member element, in the case of the member elements (i.e. it shall reference the TTCN-3 type generated for the given global XSD *element* information item).

NOTE 1: In XSD, substitution group membership is transitive, i.e. the members of a substitution group (ESG1) whose head is a member of another substitution group (ESG2) are all also members of the second substitution group (ESG2).

If the value of the head element's *abstract* attribute is "*true*", the "abstract" encoding instruction has to be attached to the field corresponding to the head element (i.e. to the first field).

NOTE 2: If the value of a member element's *abstract* attribute is "true", the "abstract" encoding instruction is attached to the TTCN-3 type generated for that element, according to clause 7.1.9.

If the head element's effective block value (see clause 7.1.10) is "*#all*" or "*substitution*", the "block" encoding instruction shall be attached to all fields of the **`union`** type except the field corresponding to the head element (the first field).

If the head element's effective block value (see clause 7.1.10) is "*restriction*" or "*extension*" the "block" encoding instruction shall be attached to all fields, generated for group member elements with a type, which has been derived from the type of the head element by *restriction* or by *extension*, respectively, at any step along the derivation path.

NOTE 3: The TTCN-3 syntax allows to attach the same attribute to several fields of the same structured type in one with attribute.

Finally, the **`union`** type shall be appended with the "untagged" encoding instruction.

NOTE 4: Declaring the head element alternative as the default alternative allows to use values and templates compatible to the head element type directly in places where values and templates of the substitution group union type are expected (see clause 6.3.2.4 in ETSI ES 201 873-1 [1]). This way, templates designed for a setting that does not use substitution groups or a version where the head element was not the head of a substitution group will still be valid when they become the head of a substitution group and this special mapping is applied.

When translating XSD references to the head element to TTCN-3, the TTCN-3 **`union`** type generated according to this clause shall be used.

EXAMPLE 1: Substitution group:

NOTE 5: Please note that the element member1 inherits its type from the head element of the substitution group.

```
<?xml version="1.0" encoding="UTF-8"?>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.example.org/SimpleCase"
            xmlns:tns="http://www.example.org/SimpleCase" >
    <!-- THE HEAD ELEMENT -->
    <xsd:element name="head" type="xsd:string" />

    <!-- SUBSTITUTION ELEMENT OF THE SAME TYPE AS THE HEAD -->
    <xsd:element name="member1" substitutionGroup="tns:head"/>

    <!-- SUBSTITUTION ELEMENT OF A TYPE RESTRICTING THE TYPE OF THE HEAD -->
    <xsd:simpleType name="stringEnum">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="something"/>
            <xsd:enumeration value="else"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:element name="member2" type="tns:stringEnum" substitutionGroup="tns:head"/>

    <!-- SUBSTITUTION ELEMENT OF A TYPE EXTENDING THE TYPE OF THE HEAD -->
    <xsd:complexType name="complexEnum">
        <xsd:simpleContent>
            <xsd:extension base="xsd:string">
                <xsd:attribute name="foo" type="xsd:float"/>
                <xsd:attribute name="bar" type="xsd:integer"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>

    <xsd:element name="member3" type="tns:complexEnum" substitutionGroup="tns:head"/>

    <!-- TOP LEVEL ELEMENT TO DEMONSTRATE SUBSTITUTION -->
    <xsd:element name="mylist">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="tns:head" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
```

```
    </xsd:element>

</xsd:schema>
```

Will be translated to TTCN-3 e.g. as:

```
module http_www_example_org_SimpleCase {
    /* SUBSTITUTION ELEMENT OF THE SAME TYPE AS THE HEAD */
    type XSD.String Member1
    with {
        variant "name as uncapitalized";
        variant "element";
    }

    /* SUBSTITUTION ELEMENT OF A TYPE RESTRICTING THE TYPE OF THE HEAD */
    type enumerated StringEnum { else_, something }
    with {
        variant "text 'else_' as 'else'";
        variant "name as uncapitalized";
    }

    type StringEnum Member2
    with {
        variant "name as uncapitalized";
        variant "element";
    }

    /* SUBSTITUTION ELEMENT OF A TYPE EXTENDING THE TYPE OF THE HEAD */
    type record ComplexEnum
    {
        XSD.Integer bar optional,
        XSD.Float foo optional,
        XSD.String base
    }
    with {
        variant "name as uncapitalized";
        variant (bar) "attribute";
        variant (foo) "attribute";
        variant (base) "untagged";
    }

    type ComplexEnum Member3
    with {
        variant "name as uncapitalized";
        variant "element";
    }

    /* THE HEAD ELEMENT */
    type union Head_group {
      @default XSD.String   head,
      Member1               member1,
      Member2               member2,
      Member3               member3
    }
    with {
        variant "untagged";
        variant(head) "element";
    }

    /* TOP LEVEL ELEMENT TO DEMONSTRATE SUBSTITUTION */
    type record Mylist
    {
        record of Head_group head_list
    }
    with {
        variant "name as uncapitalized";
        variant "element";
        variant (head_list) "untagged";
    }
} with {
    encode "XML";
    variant "namespace as 'http://www.example.org/SimpleCase' prefix 'tns'";
    variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'"
}
```

And the template:

```
template Mylist t_Ize := {
  head_list := {
    { head := "anything" },
    { member1 := "any thing" },
    { member2 := something },
    { member3 := { bar:= 5, foo := omit, base := "anything else" }
  }
}
```

Will be encoded in XML e.g. as:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:mylist
  xmlns:tns="http://www.example.org/SimpleCase"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.example.org/SimpleCase SimpleCase.xsd">
  <tns:head>anything</tns:head>
  <tns:member1>any thing</tns:member1>
  <tns:member2>something</tns:member2>
  <tns:member3>akarmi</tns:member3>
  <tns:member3 bar="5" >anything else</tns:member3>
</tns:mylist>
```

EXAMPLE 2: Effect of the block and abstract attributes on element substitution:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.example.org/BlockRestriction">

    <!-- THE HEAD ELEMENT -->
        <xsd:element name="head" type="xsd:string" block="restriction" abstract="true"/>
```

Substitution group members member1, member2, member3, their types and element "mylist" are the same as in example 1 above, hence not repeated here

```
</xsd:schema>
```

Will be translated to TTCN-3 e.g. as:

TTCN-3 type definitions Member1, StringEnum, Member2, ComplexEnum, Member3 and Mylist are the same as in example 1 above, hence not repeated here

```
module http_www_example_org_BlockRestriction {
    /* THE HEAD ELEMENT */
    type union Head_group {
      @default XSD.String   head,
      Member1               member1,
      Member2               member2,
      Member3               member3
    }
    with {
        variant "untagged";
        variant (head) "abstract";
        variant (member2) "block";
    }
```

Substitution group members member1, member2, member3, their types and element "mylist" are the same as in example 1 above, hence not repeated here

```
} with {
    encode "XML";
    variant "namespace as 'http://www.example.org/BlockRestriction' prefix 'tns'";
    variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

And the template:

```
template Mylist t_Mylist := {
  head_list := {
    { head := "anything" },
    { member1 := "any thing" },
    { member2 := something },
    { member3 := { bar:= 5, foo := omit, base := "anything else" }
  }
```

```
}
```

Can be encoded in XML e.g. as:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns: mylist
  xmlns:tns="http://www.example.org/BlockRestriction "
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.example.org/BlockRestriction BlockRestriction.xsd">

<!-- allowed to send but causes a decoding failure if present in the received XML document
    ( the head element is abstract) -->
  <tns:head>anything</tns:head>

<!-- OK to send and receive -->
  <tns:member1>any thing</tns:member1>

<!-- allowed to send but causes a decoding failure if present in the received XML document
    ( the type of member2 is derived by restriction in XSD) -->
  <tns:member2>something</tns:member2>

<!-- OK to send and receive (the type of member3 is derived by extension in XSD) -->
  <tns:member3>akarmi</tns:member3>
  <tns:member3 bar="5" >anything else</tns:member3>
</tns:mylist>
```

EXAMPLE 3: Blocking substitution:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd:="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.example.org/BlockAll"
        xmlns:tns="http://www.example.org/BlockAll">

    <!-- THE HEAD ELEMENT -->
    <xsd:element name="headNoSubstition" type="xsd:string" block="#all"/>

    <xsd:element name="groupMember1" type="xsd:string"
            substitutionGroup="tns:headNoSubstition"/>

    <xsd:element name="groupMember2" type="xsd:string"
            substitutionGroup="tns:headNoSubstition"/>

    <!-- TOP LEVEL ELEMENT TO DEMONSTRATE SUBSTITUTION -->
    <xsd:element name="mylist2">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="tns:headNoSubstition" minOccurs="0"
                                                        maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

</xsd:schema>
```

Will be translated to TTCN-3 e.g. as:

```
module http_www_example_org_BlockAll {

    type XSD.String GroupMember1
    with {
        variant "name as uncapitalized";
        variant "element";
    };

    type XSD.String GroupMember2
    with {
        variant "name as uncapitalized";
        variant "element";
    };
    /* THE HEAD ELEMENT */
    type union HeadNoSubstition_group {
      @default XSD.String   headNoSubstition,
      GroupMember1          groupMember1,
      GroupMember2          groupMember2
    }
    with {
        variant "untagged";
```

```
        variant (groupMember1, groupMember2) "block";
    }

    /* TOP LEVEL ELEMENT TO DEMONSTRATE SUBSTITUTION */
    type record Mylist2
    {
        record of HeadNoSubstition_group head_list
    }
    with {
        variant "name as uncapitalized";
        variant "element";
        variant (head_list) "untagged";
    }
} with {
  encode "XML";
  variant "namespace as 'http://www.example.org/BlockAll' prefix 'tns'";
  variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

And the template:

```
template Mylist2 t_Mylist2 := {
  head_list := {
    { headNoSubstition := "anything" },
    { groupMember1 := "any thing" },
    { groupMember2 := "something" }
  }
}
```

Can be encoded in XML e.g. as:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:mylist2
  xmlns:tns="http://www.example.org/BlockAll "
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.example.org/BlockAll BlockAll.xsd">

<!-- OK to send and receive -->
  <tns:headNoSubstition>anything</tns:headNoSubstition>

<!-- allowed to send but causes a decoding failure if present in the received XML document
    (all substitutions are disallowed) -->
  <tns:groupMember1>any thing</tns:groupMember1>

<!-- allowed to send but causes a decoding failure if present in the received XML document
    (all substitutions are disallowed) -->
  <tns:groupMember2>something</tns:groupMember2>
</tns:mylist2>
```

8.1.2 Substitution group members

XSD elements with a *substitutionGroup* attribute information item shall be translated to TTCN-3 according to clauses 7.3 and 7.1.13 with one addition: if the type of the XSD *element* is not defined in the element declaration, the type of the head element shall be used for the conversion.

8.2 Type substitution

This clause is invoked if an XSD built-in type, *simpleType* or *complexType* is referenced by the *base* attribute of the *restriction* or *extension* element information item(s) of one or more global XSD type definition(s) (i.e. the type is a parent type of one or more global derived types) AND the parent type occurs as the type of at least one XSD *element* declaration and the user has requested to generate TTCN-3 code allowing using type substitution (see clause 8). These types are called substitutable parent types (as opposed to parent types that cannot be substituted because e.g. referenced only in *attribute* declarations). Please note that when the type of an element is substituted in an instance document, XSD requires that the actual type is identified by an *xsi:type* XML attribute.

NOTE 1: This definition also includes the case when the type of an element is a built-in XSD data type and one or more user-defined types are derived from this built-in type.

In addition to the TTCN-3 types generated according to clause 7 of the present document, for each substitutable parent type a TTCN-3 **union** type shall be generated. The name of the **union** type shall be the result of applying clause 5.2.2 to the name composed of the substitutable parent type's name, or in case the parent type is a built-in XSD type the names defined in clause 6 of the present document, postfixed by "_derivations".

One alternative shall be added for the substitutable parent type itself and one alternative for each type derived from it in one or more derivation steps. The first alternative (field) of the **union** type shall be a default alternative (see clause 6.2.5 of ETSI ES 201 873-1 [1]), and it shall correspond to the substitutable parent type. The alternatives corresponding to the derived types shall be added in an ordered manner, first alphabetically ordering the types according to their target namespaces (types with no target namespace first) and subsequently alphabetically ordering the types with the same namespace based on their names. For each alternative, the field name shall be the name applying clause 5.2.2 to the name of the XSD type corresponding to the given alternative. The type of the alternative shall be:

- the TTCN-3 type generated by applying clause 7 to the substitutable parent type for the first field (corresponding to the substitutable parent type);
- the TTCN-3 type resulted by the translation of the derived type for the subsequent fields.

Finally the "useType" encoding instruction shall be attached to the TTCN-3 **union** type.

NOTE 2: Please note that the first alternative of the union is encoded without an xsi:type attribute. The user, if he wants to force xsi:type for the first alternative, needs to add the "useType" encoding instruction to the first field manually.

When translating XSD references to the substitutable parent type to TTCN-3, the TTCN-3 **union** type generated according to this clause shall be used.

NOTE 3: Declaring the substitutable parent type alternative as the default alternative allows to use values and templates compatible to the TTCN-3 type corresponding to that parent type directly in places where values and templates of the type substitution union type are expected (see clause 6.3.2.4 in ETSI ES 201 873-1 [1]). This way, templates designed for a setting that does not use type substitution or a version where the type was not the parent type of any subtype will still be valid when they become the parent of at least one subtype and this special mapping is applied.

EXAMPLE 1: Built-in type substitution:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.example.org/builtinTypeSubstitution"
  xmlns:this="http://www.example.org/builtinTypeSubstitution">

<xsd:element name="elem" type="xsd:integer"/>

<!-- derived type -->
<xsd:simpleType name="integer_deriv">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="5"/>
  </xsd:restriction>
</xsd:simpleType>
```

Will be translated to TTCN-3 e.g. as:

```
module www_example_org_builtinTypeSubstitution {

    import from XSD all;

    type Integer_derivations Elem
    with {
      variant "name as uncapitalized";
      variant "element";
    };

    <!-- derived type -->
    type XSD.Integer Integer_deriv (5 .. infinity)
    with {
      variant "name as uncapitalized";
    };

    type union Integer_derivations
    {
```

```
        @default XSD.Integer    integer_,
        Integer_deriv           integer_deriv
    }
    with {
      variant "useType";
      variant (integer_) "name as 'integer'";
    };
}
with {
  encode "XML";
  variant "namespace as 'http://www.example.org/builtinTypeSubstitution' prefix 'this'";
  variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

EXAMPLE 2: Simple type substitution:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="www.example.org/simpleTypeSubstitution"
  xmlns="www.example.org/simpleTypeSubstitution">

<xsd:element name="request" type="requestType" />

<!-- The generic type -->
<xsd:complexType name="requestType">
  <xsd:sequence>
    <xsd:element name="commonName" type="xsd:string" />
  </xsd:sequence>
</xsd:complexType>

<!-- Production specific derived type -->
<xsd:complexType name="myProductionRequestType">
  <xsd:complexContent>
    <xsd:extension base="requestType">
      <xsd:sequence>
        <xsd:element name="productionName" type="xsd:string" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<!-- Programming specific derived type -->
<xsd:complexType name="myProgrammingRequestType">
  <xsd:complexContent>
    <xsd:extension base="requestType">
      <xsd:sequence>
        <xsd:element name="programmingName" type="xsd:string" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

</xsd:schema>
```

Will be translated to TTCN-3 e.g. as:

```
module www_example_org_simpleTypeSubstitution {

    import from XSD all;

    type RequestType_derivations Request
    with {
      variant "name as uncapitalized";
      variant "element";
    };

    /* The generic type */
    type record RequestType
    {
      XSD.String commonName
    }
    with {
      variant "name as uncapitalized";
    };

    /* Production specific derived type */
    type record MyProductionRequestType
    {
```

```
      XSD.String commonName,
      XSD.String productionName
    }
    with {
      variant "name as uncapitalized";
    };

    /* Programming specific derived type */
    type record MyProgrammingRequestType
    {
      XSD.String commonName,
      XSD.String programmingName
    }
    with {
      variant "name as uncapitalized";
    };

    type union RequestType_derivations
    {
      @default RequestType      requestType,
      MyProductionRequestType   myProductionRequestType,
      MyProgrammingRequestType  myProgrammingRequestType
    }
    with {
      variant "useType";
    };

} with {
    encode "XML";
    variant "namespace as 'http://www.example.org/simpleTypeSubstitution'";
    variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
  }
```

EXAMPLE 3: Type substitution with cascaded derivations:

```
<?xml version="1.0" encoding="UTF-8"?>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="www.example.org/typeSubstCascaded3"
      xmlns="www.example.org/typeSubstCascaded3">

    <xsd:element name="request" type="requestType" />

    <!-- The generic base type -->
    <xsd:complexType name="requestType">
      <xsd:sequence>
        <xsd:element name="commonName" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>

    <!-- Production implementation -->
    <xsd:element name="product" type="myProductionRequestType" />

    <xsd:complexType name="myProductionRequestType">
      <xsd:complexContent>
        <xsd:extension base="requestType">
          <xsd:sequence>
            <xsd:element name="productionName" type="xsd:string"/>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- Derived type of myProductionRequestType -->
    <xsd:complexType name="myProductionRequestType2">
      <xsd:complexContent>
        <xsd:extension base="myProductionRequestType">
          <xsd:sequence>
            <xsd:element name="productItem" type="xsd:integer" minOccurs="0" />
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- Derived type of myProductionRequestType2 -->
    <xsd:complexType name="myProductionRequestType3">
        <xsd:complexContent>
            <xsd:restriction base="myProductionRequestType2">
                <xsd:sequence>
```

```
                <xsd:element name="commonName"  type="xsd:string" />
                <xsd:element name="productionName" type="xsd:string" />
                <xsd:element name="productItem" type="xsd:integer" minOccurs="1" />
            </xsd:sequence>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>

</xsd:schema>
```

Will be translated to TTCN-3 e.g. as:

NOTE 4: Please note that though the XSD type myProductionRequestType2 has a type derived from it, no MyProductionRequestType2_derivations **union** type is generated, because it is not used as the type of any XSD element.

```
module www_example_org_typeSubstCascaded3 {

    import from XSD all;

    type RequestType_derivations Request
    with {
      variant "name as uncapitalized";
      variant "element";
    };

    /* The generic abstract type */
    type record RequestType
    {
      XSD.String commonName
    }
    with {
      variant "name as uncapitalized";
    };

    /* Production implementation */

    type MyProductionRequestType_derivations Product
    with {
      variant "name as uncapitalized";
      variant "element";
    };

    type record MyProductionRequestType
    {
      XSD.String commonName,
      XSD.String productionName
    }
    with {
      variant "name as uncapitalized";
    };

    /* Derived type of myProductionRequestType */

    type record MyProductionRequestType2
    {
        XSD.String commonName,
        XSD.String productionName,
        XSD.Integer productItem optional
    }
    with {
        variant "name as uncapitalized";
    };


    /* Derived type of myProductionRequestType2 */

    type record MyProductionRequestType3
    {
      XSD.String commonName,
      XSD.String productionName,
      XSD.Integer productItem
    }
    with {
      variant "name as uncapitalized";
    };
```

```
    type union RequestType_derivations
    {
      @default RequestType      requestType,
      MyProductionRequestType   myProductionRequestType,
      MyProductionRequestType2  myProductionRequestType2,
      MyProductionRequestType3  myProductionRequestType3,
    }
    with {
    variant "useType";
    };


    type union MyProductionRequestType_derivations
    {
      @default MyProductionRequestType  myProductionRequestType,
      MyProductionRequestType2          myProductionRequestType2,
      MyProductionRequestType3          myProductionRequestType2,
    }
    with {
    variant "useType";
    };

}
with {
    encode "XML";
    variant "namespace as 'http://www.example.org/typeSubstCascaded3'";
    variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

If the value of the substitutable parent type's *abstract* attribute is "*true*", the "abstract" encoding instruction has to be attached to the field corresponding to the substitutable parent type, i.e. to the first field.

NOTE 5: If the value of a derived type's *abstract* attribute is "true", the "abstract" encoding instruction is attached to the TTCN-3 type generated for that XSD type, according to clause 7.1.9.

If the substitutable parent type's effective block value (see clause 7.1.10) is "*#all*", the "block" encoding instruction shall be attached to all fields of the **union** type except the field corresponding to the substitutable parent type (the first field).

If the substitutable parent type's effective block value (see clause 7.1.10) is "*restriction*" or "*extension*" the "block" encoding instruction shall be attached to all fields, generated for types, derived from the substitutable parent type by *restriction* or by *extension*, respectively, at any step along the derivation path.

NOTE 6: The TTCN-3 syntax allows to attach the same attribute to several fields of the same structured type in one with attribute.

EXAMPLE 4: Mapping a substitutable abstract type:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="www.example.org/typeSubstitutionAbstract"
  xmlns="www.example.org/typeSubstitutionAbstract">

<xsd:element name="request" type="requestAbstractType" />

<!-- The generic abstract type -->
<xsd:complexType name="requestAbstractType" abstract="true">
  <xsd:sequence>
    <xsd:element name="commonName" type="xsd:string" />
  </xsd:sequence>
</xsd:complexType>

<!-- Production implementation -->
<xsd:complexType name="myProductionRequestType">
  <xsd:complexContent>
    <xsd:extension base="requestAbstractType">
      <xsd:sequence>
        <xsd:element name="productionName" type="xsd:string" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<!-- Programming implementation -->
```

```
<xsd:complexType name="myProgrammingRequestType">
  <xsd:complexContent>
    <xsd:extension base="requestAbstractType">
      <xsd:sequence>
        <xsd:element name="programmingName" type="xsd:string" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

</xsd:schema>
```

Will be translated to TTCN-3 e.g. as:

```
module www_example_org_typeSubstitutionAbstract {

    import from XSD all;

    type RequestAbstractType_derivations Request
    with {
    variant "name as uncapitalized";
    variant "element";
    };

    /* The generic abstract type */
    type record RequestAbstractType
    {
        XSD.String commonName
    }
    with {
      variant "name as uncapitalized";
      variant "abstract";
    };

    /* Production implementation */

    type record MyProductionRequestType
    {
        XSD.String commonName,
        XSD.String productionName
    }
    with {
      variant "name as uncapitalized";
    };

    /* Programming implementation */
    type record MyProgrammingRequestType
    {
      XSD.String commonName,
      XSD.String programmingName
    }
    with {
      variant "name as uncapitalized";
    };

    type union RequestAbstractType_derivations
    {
        @default RequestAbstractType    requestAbstractType,
        MyProductionRequestType myProductionRequestType,
        MyProgrammingRequestType    myProgrammingRequestType
    }
    with {
      variant "useType";
      variant (requestAbstractType) "abstract";
    };
} with {
    encode "XML";
    variant "namespace as 'http://www.example.org/typeSubstitutionAbstract'";
    variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

Annex A (normative): TTCN-3 module XSD

This annex defines a TTCN-3 module containing type definitions equivalent to XSD built-in types.

NOTE: The capitalized type names used in annex A of Recommendation ITU-T X.694 [4] have been retained for compatibility. All translated structures are the result of two subsequent transformations applied to the XSD Schema: first, transformations described in Recommendation ITU-T X.694 [4], then transformations described in ETSI ES 201 873-7 [i.11]. In addition, specific extensions are used that allow codecs to keep track of the original XSD nature of a given TTCN-3 type.

```
module XSD {

//These constants are used in the XSD date/time type definitions
const charstring
  dash := "-",
  cln  := ":",
  year := "[0-9]#(4)",
  yearExpansion := "-#(,1)([1-9][0-9]#(0,))#(,1)",
  month := "(0[1-9]|1[0-2])",
  dayOfMonth := "(0[1-9]|[12][0-9]|3[01])",
  hour := "([01][0-9]|2[0-3])",
  minute := "([0-5][0-9])",
  second := "([0-5][0-9])",
  sFraction := "(.[0-9]#(1,))#(,1)",
  endOfDayExt := "24:00:00(.0#(1,))#(,1)",
  nums := "[0-9]#(1,)",
  ZorTimeZoneExt := "(Z|[\+\-]((0[0-9]|1[0-3]):[0-5][0-9]|14:00))#(,1)",
  durTime := "(T[0-9]#(1,)"&
             "(H([0-9]#(1,)(M([0-9]#(1,)(S|.[0-9]#(1,)S))#(,1)|.[0-9]#(1,)S|S))#(,1)|"&
             "M([0-9]#(1,)(S|.[0-9]#(1,)S)|.[0-9]#(1,)M)#(,1)|"&
             "S|"&
             ".[0-9]#(1,)S))";

//anySimpleType

type XMLCompatibleString AnySimpleType with {

  variant "XSD:anySimpleType";
}
//anyType;

type record AnyType
{
  record of XSD.String embed_values optional
  record length (1 .. infinity) of String attr optional,
  record of String elem_list
} with {
     variant "XSD:anyType";
     variant (attr) "anyAttributes";
     variant (elem_list) "anyElement";
}

type record AnyComplexType {
    record of XSD.String embed_values optional,
    record length (1 .. infinity) of anytype attr optional,
    record of anytype elem_list
}
with {
    variant "XSD:anyType";
    variant "embedValues";
    variant(attr) "anyAttributes";
    variant(elem_list) "anyElement";
}


// String types

  type  XMLCompatibleString String with {
         variant "XSD:string";
    }

  type  XMLStringWithNoCRLFHT NormalizedString  with {
         variant "XSD:normalizedString";
```

```
  }

  type  NormalizedString Token  with {
        variant "XSD:token";
  }

  type  XMLStringWithNoWhitespace Name  with {
        variant "XSD:Name";
  }

  type  XMLStringWithNoWhitespace NMTOKEN  with {
        variant "XSD:NMTOKEN";
  }

  type  Name NCName  with {
        variant "XSD:NCName";
  }

  type  NCName ID  with {
        variant "XSD:ID";
  }

  type  NCName IDREF  with {
        variant "XSD:IDREF";
  }

  type  NCName ENTITY  with {
        variant "XSD:ENTITY";
  }

  type octetstring  HexBinary with {
        variant "XSD:hexBinary";
  }

  type octetstring Base64Binary with {
       variant "XSD:base64Binary";
  }

  type XMLStringWithNoCRLFHT AnyURI with {
        variant "XSD:anyURI";
  }

  type charstring  Language (pattern "[a-zA-Z]#(1,8)(-\w#(1,8))#(0,)") with {
        variant "XSD:language";
  }


// Integer types

  type integer  Integer with {
        variant "XSD:integer";
  }

  type integer  PositiveInteger (1 .. infinity) with {
        variant "XSD:positiveInteger";
  }

  type integer  NonPositiveInteger (-infinity .. 0) with {
        variant "XSD:nonPositiveInteger";
  }

  type integer  NegativeInteger (-infinity .. -1) with {
        variant "XSD:negativeInteger";
  }

  type integer  NonNegativeInteger (0 .. infinity) with {
        variant "XSD:nonNegativeInteger";
  }

  type longlong  Long with {
        variant "XSD:long";
  }

  type unsignedlonglong  UnsignedLong with {
        variant "XSD:unsignedLong";
  }
```

```
  type long  Int with {
        variant "XSD:int";
    }

  type unsignedlong  UnsignedInt with {
        variant "XSD:unsignedInt";
    }

  type short  Short with {
        variant "XSD:short";
    }

  type unsignedshort  UnsignedShort with {
        variant "XSD:unsignedShort";
    }

  type byte  Byte with {
        variant "XSD:byte";
    }

  type unsignedbyte  UnsignedByte with {
        variant "XSD:unsignedByte";
    }

// Float types

  type float  Decimal (!-infinity .. !infinity) with {
        variant "XSD:decimal";
    }

  type IEEE754float  Float with {
        variant "XSD:float";
    }

  type IEEE754double  Double with {
        variant "XSD:double";
    }

// Time types

type charstring  Duration (pattern
  "{dash}#(,1)P({nums}(Y({nums}(M({nums}D{durTime}#(,1)|{durTime}#(,1))|D{durTime}#(,1))|" &
  "{durTime}#(,1))|M({nums}D{durTime}#(,1)|{durTime}#(,1))|D{durTime}#(,1))|{durTime})"
  ) with {
        variant "XSD:duration";
    }

type charstring  DateTime (pattern
  "{yearExpansion}{year}{dash}{month}{dash}{dayOfMonth}T({hour}{cln}{minute}{cln}{second}" &
  "{sFraction}|{endOfDayExt}){ZorTimeZoneExt}"
  ) with {
        variant "XSD:dateTime";
    }

type charstring  Time (pattern
  "({hour}{cln}{minute}{cln}{second}{sFraction}|{endOfDayExt}){ZorTimeZoneExt}"
  ) with {
        variant "XSD:time";
    }

type charstring  Date (pattern
  "{yearExpansion}{year}{dash}{month}{dash}{dayOfMonth}{ZorTimeZoneExt}"
  ) with {
        variant "XSD:date";
    }

type charstring  GYearMonth (pattern
  "{yearExpansion}{year}{dash}{month}{ZorTimeZoneExt}"
  ) with {
        variant "XSD:gYearMonth";
    }

type charstring  GYear (pattern
  "{yearExpansion}{year}{ZorTimeZoneExt}"
  ) with {
        variant "XSD:gYear";
    }
```

```
type charstring  GMonthDay (pattern
  "{dash}{dash}{month}{dash}{dayOfMonth}{ZorTimeZoneExt}"
  ) with {
        variant "XSD:gMonthDay";
    }

type charstring  GDay (pattern
  "{dash}{dash}{dash}{dayOfMonth}{ZorTimeZoneExt}"
  ) with {
        variant "XSD:gDay";
    }

type charstring  GMonth (pattern
  "{dash}{dash}{month}{ZorTimeZoneExt}"
  ) with {
        variant "XSD:gMonth";
    }


// Sequence types

    type  record of NMTOKEN NMTOKENS with {
        variant "XSD:NMTOKENS";
    }

    type  record of IDREF IDREFS with {
        variant "XSD:IDREFS";
    }

    type  record of ENTITY ENTITIES with {
        variant "XSD:ENTITIES";
    }


    type record QName
    {
      AnyURI uri  optional,
      NCName name
    } with {
         variant "XSD:QName";
    }

// Boolean type

    type boolean  Boolean with {
        variant "XSD:boolean";
    }

//TTCN-3 type definitions supporting the mapping of W3C XML Schema built-in datatypes


type utf8string XMLCompatibleString
 (
  char(0,0,0,9).. char(0,0,0,9),
  char(0,0,0,10)..char(0,0,0,10),
  char(0,0,0,13)..char(0,0,0,13),
  char(0,0,0,32)..char(0,0,215,255),
  char(0,0,224,0)..char(0,0,255,253),
  char(0,1,0,0)..char(0,16,255,253)
);

type utf8string XMLStringWithNoWhitespace
 (
  char(0,0,0,33)..char(0,0,215,255),
  char(0,0,224,0)..char(0,0,255,253),
  char(0,1,0,0)..char(0,16,255,253)
);

type utf8string XMLStringWithNoCRLFHT
 (
  char(0,0,0,32)..char(0,0,215,255),
  char(0,0,224,0)..char(0,0,255,253),
  char(0,1,0,0)..char(0,16,255,253)
);

}//end module
```

Annex B (normative): Encoding instructions

B.0 General

As described in clause 5 of the present document, in case of explicit mapping, the information not necessary to produce valid TTCN-3 abstract types and values but needed to produce the correct encoded value (an XML document), shall be retained in encoding instructions. Encoding instructions are contained in TTCN-3 **`encode`** and **`variant`** attributes associated with the TTCN-3 definition, field or value of a definition. This annex defines the encoding instructions for the XSD to TTCN-3 mapping. Clause B.2.2 defines special overwriting rules for variants for XML encoding.

NOTE 1: In case of implicit mapping the information needed for correct encoding is to be retained by the TTCN-3 tool internally and thus its form is out of scope of the present document.

If templates or values of types with the "XML" **`encode`** attribute (or any of its synonyms - i.e. TTCN-3 types that are a result of conversion from XSD) are used as an argument of a communication operation, a **`variant`** attribute containing the "element" encoding instruction shall be assigned to it. Using a value or template of a type that does not contain this encoding instruction shall cause an error.

NOTE 2: Valid XML documents can be based only on XSD element definitions. This rule allows TTCN-3 tools to statically check that a correct TTCN-3 type is used for sending or receiving XML content.

If templates or values of types with the "XML" **`encode`** attribute (or any of its synonyms - i.e. TTCN-3 types that are a result of conversion from XSD) are used as parameters of the predefined codec functions, the **`variant`** attribute shall contain either the "element" or "attribute" encoding instruction or the parameter shall be a field of a record or union to which - at the top level - the "element" encoding instruction is assigned. Using a value or template of a type that does not contain one of these encoding instructions shall cause an error.

NOTE 3: Fragments of XML documents can be based only on XSD element or attribute definitions. This rule allows TTCN-3 tools to statically check that a correct TTCN-3 type is used when explicitly encoding or decoding fragments of XML documents (e.g. used in any element of anyAttributes fields).

B.1 General

A single attribute shall contain one encoding instruction only. Therefore, if several encoding instructions shall be attached to a TTCN-3 language element, several TTCN-3 attributes shall be used.

The "syntactical structure" paragraphs of each clause below identify the syntactical elements of the attribute (i.e. inside the "with { }" statement). The syntactical elements shall be separated by one or more whitespace characters.
A syntactical element may precede or follow a double quote character without a whitespace character. There shall be no whitespace between an opening single quote character and syntactical element directly following it and between a closing single quote character and the syntactical element directly preceding it. All characters (including whitespaces) between a pair of single quote characters shall be part of the encoding instruction.

Typographical conventions: **bold** font identify TTCN-3 keywords. The syntactical elements *freetext* and *name* are identified by *italic* font; they shall contain one or more characters and their contents are specified by the textual description of the encoding instruction. Normal font identify syntactical elements that shall occur within the TTCN-3 attribute as appear in the syntactical structure. The following character sequences identify syntactical rules and shall not appear in the encoding instruction itself:

- (|) - identify alternatives.
- [ ] - identify that the part of the encoding instruction within the square brackets is optional.
- { } - identify zero or more occurrences of the part between the curly brackets.
- """ - identify the opening or the enclosing double quote of the encoding instruction.

B.2 Basic XML encode and variant attribute rules

B.2.1 The XML encode attribute

The encode attribute "XML" shall be used to identify that the definitions in the scope unit to which this attribute is attached shall be encoded in one of the following XML formats:

- "XML" or "XML1.0" for W3C® XML 1.0; and
- "XML1.1" for W3C® XML 1.1.

```
Syntactical structure
    encode """ (XML | XML1.0 | XML1.1 ) """
Applicable to (TTCN-3)
Module, group, definition.
```

B.2.2 Variant Attribute Overwriting Rules

In the context of XML encoding the following special rules apply for overwriting **variant** attributes.

Each module, group, type or field in a TTCN-3 module has a set of variant attributes *VA* that are attached to it. This set is a combination of three sets using the **override** operator as defined below: the context variant attributes *CA* inherited from the container, the definition variant attributes *DA* added in the definition and the referenced variant attributes *VA* of the referenced type if the definition references another type. The resulting *composed* variant attribute is a concatenation of the elements of the variant attribute set interspersed with semicolons.

Each variant attribute has the general form """*VariantKind*[*VariantValue*]""" where the *VariantKind* is the first word in the attribute and *VariantValue* is the rest of the attribute for those attributes that are not single-word attributes. To describe the relationships below, the sets of variant attribute tuples (*VariantKind*, *VariantValue*) are used.

The **override** operator is defined by the following set of transformations where names in uppercase cursive are variables , names in bold are actual values and *A* **|** *B* is the set union of sets *A* and *B* with the union operator **|** having higher precedence than the **override** operator. The left-hand sides of the transformations are matched to the operands in descending order, the first one that matches is applied.

- { (**transparent**, *Value1*) } **|** *V1* **override** *V2* => { (**transparent**, *Value1*) } **|** (*V1* **override** *V2*)
  The transparent attributes within the overriding attributes, will be kept without change.
- { (*Kind*, *Value1*) } **|** *V1* **override** { (*Kind*, *Value2*) } **|** *V2* => { (*Kind*, *Value1*) } **|** *V1* **override** *V2*
  Non-transparent attributes in the overriding set, override all attributes with the same kind.
- *V1* **override** *V2* => *V1* **|** *V2*
  When none of the above apply the result is the union of attributes from both sets.

The set of variant attributes definition added by the *DA(E)* for a defined entity E are those variant attributes directly attached to the definition of *E*. For modules, groups and types, those are the variant attributes attached to their definition without referencing a field of the definition. For fields of a type, those are the variant attributes attached to the enclosing type definition for that field.

The only context-inheritable variant attributes are those of *VariantKind* **attributeFormQualified**, **elementFormQualified**, **namespace** and **controlNamespace**. Let *E* be a definition and *P* its parent, then the set CA(E) of context attributes can be defined as follows (where *CA(P)* is the empty set if *E* is a module):

- *CA(E)* == { (*K*, *V*) from (*DA(E)* **override** *CA(P)*) where *K* is context-inheritable variantkind}
  | { (**name**, **as** *X*) where *DA(P)* contains (**name**, **all as** *X*) }

Thus, the context attributes of an entity are those of its definition overriding the ones from its container, plus the **name as** attribute, if the parent defines a **name all as** attribute. The latter subset is not transitively inherited as the **name** attribute is not context-inheritable.

Finally, the set of variant attributes attached to an entity *E* referencing a type *T* is defined as follows:

- *VA(E) == VA(T)* **override** *DA(E)* **override** *CA(E)*

Thus, the variant attributes of the referenced type override the variant attributes attached directly to the definition of entity *E* which again override the variant attributes inherited from the context of *E*. If the definition *E* does not reference a type or the referenced type is a builtin type, *VA(T)* is the empty set.

B.3 Encoding instructions

B.3.1 XSD data type identification

Syntactical structure(s)

variant """ (XSD:string | XSD:normalizedString | XSD:token | XSD:Name | XSD:NMTOKEN | XSD:NCName | XSD:ID | XSD:IDREF | XSD:ENTITY | XSD:hexBinary | XSD:base64Binary | XSD:anyURI | XSD:language | XSD:integer | XSD:positiveInteger | XSD:nonPositiveInteger | XSD:negativeInteger | XSD:nonNegativeInteger | XSD:long | XSD:unsignedLong | XSD:int | XSD:unsignedInt | XSD:short | XSD:unsignedShort | XSD:byte | XSD:unsignedByte | XSD:decimal | XSD:float | XSD:double | XSD:duration | XSD:dateTime | XSD:time | XSD:date | XSD:gYearMonth | XSD:gYear | XSD:gMonthDay | XSD:gDay | XSD:gMonth | XSD:NMTOKENS | XSD:IDREFS | XSD:ENTITIES | XSD:QName | XSD:boolean) """

Applicable to (TTCN-3)

These encoding instructions shall not appear in a TTCN-3 module mapped from XSD. They are attached to the TTCN-3 type definitions corresponding to XSD data types.

Description

The encoder and decoder shall handle instances of a type according to the corresponding XSD data type definition. In particular, record of elements of instances corresponding to the XSD sequence types *NMTOKENS IDREFS* and *ENTITIES* shall be combined into a single XML list value using a single space as separator between the list elements. At decoding the XML list value shall be mapped to a TTCN-3 record of value by separating the list into its itemType elements (the whitespaces between the itemType elements shall not be part of the TTCN-3 value). The `uri` and `name` fields of a TTCN-3 instance of an XSD:QName type shall be combined to an XSD QName value at encoding. At decoding an XSD QName value shall be separated to the URI part and the non-qualified name part (the double colon between the two shall be disposed) and those parts shall be assigned to the `uri` and `name` fields of the corresponding TTCN-3 value correspondingly.

B.3.2 Any element

Syntactical structure(s)

variant """ anyElement [except ('*freetext*' | unqualified) | from [unqualified ,] [{ '*freetext*' , } '*freetext*']] """

Applicable to (TTCN-3)

Fields of structured types generated for the XSD *any* element (see clause 7.7.1).

NOTE 1: If the *any* element has a maxOccurs attribute with a value more than 1 (including "unbounded"), the element is mapped to a **`record of`** `XSD.String` field, in which case the anyElement instruction will be applied to the XSD.String type as well, as in all other cases. See for example the conversion of XSD complex type e46b in clause 7.7.1.

Description

One TTCN-3 encoding instruction shall be generated for each field corresponding to an XSD *any* element. The *freetext* part(s) shall contain the URI(s) identified by the *namespace* attribute of the XSD *any* element. The *namespace* attribute may also contain wildcards. They shall be mapped as given in table B.1.

Table B.1: Mapping namespace attribute wildcards

Facet / type	Value of the XSD namespace attribute	"except" or "from" clause in the TTCN-3 attribute	Remark
	##any	*<nor except neither from clause present>*	
	##local	from unqualified	
##other		except unqualified, "*<target namespace of the ancestor schema element of the given any element>"*	Also disallows unqualified elements, i.e. elements without a target namespace
##other		except unqualified	In the case no target namespace is ancestor schema element of the given any element
##targetNamespace		from "*<target namespace of the ancestor schema element of the given any element >"*	
"http://www.w3.org/1999/xhtml ##targetNamespace"		from "http://www.w3.org/1999/xhtml", "*<target namespace of the ancestor schema element of the given any element >"*	

In the encoding process the content of the TTCN-3 value shall be handled transparently, except when maxOccurs is greater than 1: in this case the elements of the TTCN-3 **record of** value (corresponding to the *any* XSD element), shall be concatenated transparently to produce the encoded XML value. Transparency in this case means that if the element instance contains an *xmlns* attribute, it shall be not be checked or changed by the encoder. If the element instance does not contain an *xmlns* attribute, but the TTCN-3 type or field defines a namespace, except using the wildcards ##any, ##local or ##other, the *xmlns* attribute with the value of the namespace shall be added by the encoder.

In the decoding process, the decoder shall check if the fragment of the received XML document corresponding to the TTCN-3 field with the "anyElement" encoding instruction fulfils the namespace specification in the encoding instruction and, if no "processContents" encoding instruction is present for the element being decoded, it shall check if it is a well-formed XML element (i.e. the content shall be assessed according to XML Schema Part 1 [9], clause 3.10.1, assessment level *skip*. The default value of namespace specification is "##any". If a "processContents" encoding instruction is present, the content shall be assessed according to it. The failure of the namespace checking or the content assessment shall cause a decoding failure.

NOTE 2: Please note that any other assessment level (*strict* or *lax*) could result in different outcomes if a schema related to the content of the *any* element is available for the decoder or not. As this would have adverse effect on test result reproducibility, only the *skip* assessment level is necessary.

The name of the TTCN-3 field containing the *anyElement* instruction has no effect on the encoding or decoding process and it is ignored by the codec.

B.3.3 Any attributes

Syntactical structure(s)

variant """ anyAttributes [except '*freetext*' | from [unqualified ,] { '*freetext*', } '*freetext*'] """

Applicable to (TTCN-3)

Fields of structured types generated for the XSD *anyAttribute* element (see clause 7.7.2).

Description

One TTCN-3 encoding instruction shall be generated for each field corresponding to an XSD *anyAttribute* element. The *freetext* part(s) shall contain the URI(s) identified by the *namespace* attribute of the XSD *anyAttribute* element. The *namespace* attribute may also contain wildcards. They shall be mapped as given in table B.1.

In the encoding process, if the type is encoded as a top-level type, this encoding instruction shall be ignored.

In all other cases, in the encoding process one XML attribute shall be added to the XML element being encoded for each element of the corresponding TTCN-3 record of value. When the <*URI*> part is present in the given TTCN-3 string element (see clause 7.7.2), the encoder shall use the <*URI*> and the <*non-qualified attribute name*> part of string to create a qualified XML attribute name and, using the <*attribute value*> part it shall create a valid XML attribute. When the <*URI*> part is not present, the XML attribute created for the given record of element shall have a non-qualified name in the XML instance. See also example in clause 7.7.2. The order of the generated XML attribute shall correspond to the order they are defined in the record of value to which the encoding instruction relates to. The namespace prefix used and if already existing namespace prefixes identifying a given namespace is reused or not, is an encoder option.

In the decoding process, the decoder shall create one TTCN-3 record of element for each attribute of the XML element being decoded that is not from the control namespace, and whose name is not that of the identifier (possibly modified in accordance with any final "name as" or "namespace as" encoding instructions) of another component of the enclosing type that has a final "attribute" encoding instruction. The decoder shall create the TTCN-3 strings (the elements of the record of to which the "anyAttribute" encoding instruction is attached) in the order of the affected XML attributes in the XML element. The decoder shall check if the namespace of the actually decoded XML attribute satisfies the namespace restrictions of the "anyAttribute" encoding instruction (including the no namespace case) and in case of non-compliance it shall cause a decoding failure. If the XML attribute has a namespace-qualified name, the <*URI*> part (see clause 7.7.2) of the generated string value shall be present, otherwise the <*URI*> part shall be absent. If the <*URI*> part present, the decoder shall insert a lonely SPACE character between the <*URI*> and the <*non-qualified attribute name*> parts of the generated TTCN-3 string value.
The name of the TTCN-3 field containing the *anyAttributes* instruction has no effect on the encoding or decoding process and it is ignored by the codec.

B.3.4 Attribute

Syntactical structure(s)

variant """ attribute """

Applicable to (TTCN-3)

Top-level type definitions and fields of structured types generated for XSD *attribute* elements.

Description

This encoding instruction designates that the instances of the TTCN-3 type or field shall be encoded and decoded as XML attributes.

B.3.5 AttributeFormQualified

Syntactical structure(s)

variant """ attributeFormQualified """

Applicable to (TTCN-3)

Modules.

Description

This encoding instruction designates that names of XML attributes that are instances of TTCN-3 definitions in the given module shall be encoded as qualified names and at decoding qualified names shall be expected as valid attribute names.

B.3.6 Control namespace identification

Syntactical structure(s)

variant """ controlNamespace '*freetext*' prefix '*freetext*' """

Applicable to (TTCN-3)

Module.

Description

This encoding instruction commands the encoder to use the identified namespace and prefix whenever a `type`, *nil*, *schemalocation* or ***noNamespaceSchemaLocation*** schema-related attributes are to be inserted into the encoded XML document (see also clauses 3.1 and 5.1.5 of the present document). The first *freetext* component shall identify a syntactically valid namespace and the second *freetext* component shall identify a namespace prefix.

B.3.7 Default for empty

Syntactical structure(s)

variant """ defaultForEmpty as '*freetext*' """

Applicable to (TTCN-3)

TTCN-3 components generated for XSD *attribute* or *element* elements with a *fixed* or *default* attribute.

Description

The "*freetext*" component shall designate a valid value of the type to which the encoding instruction is attached to.

This encoding instruction has no effect on the encoding process and designates that the decoder shall insert the value specified by *freetext* if the corresponding attribute is omitted or when the corresponding element appears without any content in the XML instance being decoded; it has no effect in other cases.

NOTE: If an element with a defaultForEmpty encoding instruction attached is missing in the XML instance being decoded, its corresponding field will also be absent in the decoded TTCN-3 value.

B.3.8 Element

Syntactical structure(s)

variant """ element """

Applicable to (TTCN-3)

Top-level type definitions generated for XSD *element* elements that are direct children of a *schema* element.

Description

This encoding instruction designates that the instances of the TTCN-3 type shall be encoded and decoded as XML elements.

B.3.9 ElementFormQualified

Syntactical structure(s)

variant """ elementFormQualified """

Applicable to (TTCN-3)

Modules.

Description

This encoding instruction designates that tags of XML local elements that are instances of TTCN-3 definitions in the given module shall be encoded as qualified names and at decoding qualified names shall be expected as valid element tags names.

B.3.10 Embed values

Syntactical structure(s)

variant """ embedValues """

Applicable to (TTCN-3)

TTCN-3 record types generated for XSD *complexType*-s and *complexContent*-s with the value of the *mixed* attribute "true".

Description

The encoder shall encode the record type to which this attribute is applied in a way, which produces the same result as the following procedure: first a partial encoding of the record is produced, ignoring the embed_values field. The first string of the embed_values field (the first record of element) shall be inserted at the beginning of the partial encoding, before the start-tag of the first XML element (if any). Each subsequent string shall be inserted between the end-tag of the XML element and the start-tag of the next XML element (if any), until all strings are inserted. In the case the maximum allowed number of strings is present in the TTCN-3 value (the number of the XML elements in the partial encoding plus one) the last string will be inserted after end-tag of the last element (to the very end of the partial encoding). The following special cases apply:

a) At decoding, strings before, in-between and following the XML elements shall be collected as individual components of the embed_values field. If no XML elements are present, and there is also a defaultForEmpty encoding instruction on the sequence type, and the encoding is empty, a decoder shall interpret it as an encoding for the *freetext* part specified in the defaultForEmpty encoding instruction and assign this abstract value to the first (and only) component of the embed_values field.

b) If the type also has a useNil encoding instruction and the optional component is absent, then the embedValues encoding instruction has no effect.

c) If the type has a useNil encoding instruction and if a decoder determines that the optional component is present, by the absence of a nil identification attribute (or its presence with the value false), then item a) above shall apply.

B.3.11 Form

Syntactical structure(s)

variant """ form as (qualified | unqualified) """

Applicable to (TTCN-3)

Top-level type definitions generated for XSD *attribute* elements and fields of structured type definitions generated for XSD *attribute* or *element* elements.

Description

This encoding instruction designates that names of XML attributes or tags of XML local elements corresponding to instances of the TTCN-3 type or field of type to which the form encoding instruction is attached, shall be encoded as qualified or unqualified names respectively and at decoding qualified or unqualified names shall be expected respectively as valid attribute names or element tags.

B.3.12 List

Syntactical structure(s)

variant """ list """

Applicable to (TTCN-3)

Record of types mapped from XSD *simpleType*-s derived as a list type.

Description

This encoding instruction designates that the record of type shall be handled as an XSD list type, namely, record of elements of instances shall be combined into a single XML list value using a single SP(32) (space) character as separator between the list elements. At decoding the XML list value shall be mapped to a TTCN-3 record of value by separating the list into its itemType elements (the whitespaces between the itemType elements shall not be part of the TTCN-3 value).

B.3.13 Name as

Syntactical structure(s)

variant """ name (as ('*freetext*' | changeCase) | all as changeCase) """,

where changeCase := (capitalized | uncapitalized | lowercased | uppercased)

Applicable to (TTCN-3)

Type or field of structured type. The form when *freetext* is empty shall be applied to fields of union types with the "useUnion" encoding instruction only (see clause B.3.16).

Description

The name encoding instruction identifies if the name of the TTCN-3 definition or field differs from the value of the *name* attribute of the related XSD element. The name resulted from applying the name encoding attribute shall be used as the non-qualified part of the name of the corresponding XML attribute or element tag.

When the "name as '*freetext*'" form is used, *freetext* shall be used as the attribute name or element tag, instead of the name of the related TTCN-3 definition (e.g. TTCN-3 type name or field name).

The "name as "" (i.e. freetext is empty) form designates that the TTCN-3 field corresponds to an XSD unnamed type, thus its name shall not be used when encoding and decoding XML documents.

The "name as capitalized" and "name as uncapitalized" forms identify that only the first character of the related TTCN-3 type or field name shall be changed to lower case or upper case respectively.

The "name as lowercased" and "name as uppercased" forms identify that each character of the related TTCN-3 type or field name shall be changed to lower case or upper case respectively.

The "name all as capitalized", "name all as uncapitalized", "name as lowercased" and "name as uppercased" forms has effect on all direct fields of the TTCN-3 definition to which the encoding instruction is applied (e.g. in case of a structured type definition to the names of its fields in a non-recursive way but not to the name of the definition itself and not to the name of fields embedded to other fields).

The **`name`** encoding instruction shall not be applied when the **`untagged`** encoding instruction is used. However, if both instructions are applied to the same TTCN-3 component in the same or in different TTCN-3 definitions, the **`untagged`** instruction takes precedence (i.e. no start and end tags shall be used, see clause B.3.21).

B.3.14 Namespace identification

Syntactical structure(s)

variant """ namespace as '*freetext*' [prefix '*freetext*'] """

Applicable to (TTCN-3)

- Modules.
- Fields of record types generated for *attribute*s of *complexTypes* taken in to *complexType* definitions by referencing *attributeGroup*(s), defined in *schema* elements with a different (but not absent) target namespace and imported into the *schema* element which is the ancestor of the *complexType*.

Description

The first *freetext* component identifies the namespace to be used in qualified XML attribute names and element tags at encoding, and to be expected in received XML documents. The second *freetext* component is optional and identifies the namespace prefix to be used at XML encoding. If the prefix is not specified, the encoder shall either identify the namespace as the default namespace (if all other namespaces involved in encoding the XML document have prefixes) or shall allocate a prefix to the namespace (if more than one namespace encoding instructions are missing the prefix part).

B.3.15 Nillable elements

Syntactical structure(s)

variant """ useNil """

Applicable to (TTCN-3)

Top-level record types or record fields generated for nillable XSD *element* elements.

Description

The encoding instruction designates that the encoder, when the optional TTCN-3 field of the **`record`** added for the nillable element with the name clause 5.2.2 applied to "content" (see clause 7.1.11) is omitted, it shall produce the XML element with the xsi:nil="true" attribute and no value.

When the nillable XML element is present in the received XML document and carries the xsi:nil="true" attribute, the above optional field of the record in the corresponding TTCN-3 value shall be set to **omit**. If the nillable XML element carries the xsi:nil="true" attribute and has a children (either any character or element information item) at the same time, the decoder shall produce an error.

NOTE: In case of **`record`**s generated for nillable *complexType elements* , the first field of the outer record without the "attribute" encoding instruction corresponds to the value (content) of that *element*.

B.3.16 Use union

Syntactical structure(s)

variant """ useUnion """

Applicable to (TTCN-3)

Types and field of TTCN-3 **union** types generated for XSD *simpleTypes* derived by *union* without restriction on enumerated values, and to types and fields of TTCN-3 **record** types generated for XSD *simpleTypes* derived by *union* restricted on enumerated values (see clause 7.5.3).

Description

When used on a TTCN-3 **union** type (generated for XSD *simpleTypes* derived by *union* without restriction on enumerated values): the encoding instruction designates that the encoder shall not use the start-tag and the end-tag around the encoding of the selected alternative (field of the TTCN-3 union type) and shall use the type identification attribute (xsi:type), identifying the XSD base datatype of the selected alternative, except when encoding attributes or the encoded component has a "list" encoding instruction attached or the "noType" encoding instruction is also present (see clause B.3.27). At decoding the decoder shall place the received XML value into the corresponding alternative of the TTCN-3 **`union`** type, based on the received value and the type identification attribute, if present.

When used on a TTCN-3 **record** type (generated for XSD *simpleTypes* derived by *union* restricted on enumerated values): the encoding instruction designates that the encoder shall not use the start-tag and the end-tag around the encoding of the selected alternative (the "content" field of the TTCN-3 **record** type) and shall use the type identification attribute (xsi:type), identifying the XSD base datatype of the selected alternative (the "xsiType" field of the TTCN-3 **record** type), except when encoding attributes or the encoded component has a "list" encoding instruction attached or the "noType" encoding instruction is also present (see clause B.3.27). At decoding the received XML value shall be validated against the member types (represented in the "xsiType" field of the TTCN-3 **record** type) and values (represented in the "content" field of the TTCN-3 **record** type) separately, in the order in which they appear in the XSD definition until a match is found.

NOTE: The decoding procedure allows multiple combinations for enumerated values from the XSD restriction (see clause 7.5.3). E.g. in example 5 in clause 7.5.3 the enumerated value "x20" may indicate a **float** or **integer** value, and should be resolved to the first in order of appearance in the XSD type.

B.3.17 Text

Syntactical structure(s)

variant """ text ('*name*' as ('*freetext*' |) | all as changeCase) """

NOTE 1: The definition of changeCase is given in clause B.3.13.

Applicable to (TTCN-3)

Enumeration types generated for XSD enumeration facets where the enumeration base is a string type (see clause 6.1.5, first paragraph), and the name(s) of one or more TTCN-3 enumeration values is(are) differs from the related XSD enumeration item. XSD.Boolean types, instances of XSD.Boolean types (see clause 6.7).

Description

When *name* is used, it shall be generated for the differing enumerated values only. The *name* shall be the identifier of the TTCN-3 enumerated value the given instruction relates to. If the difference is that the first character of the XSD enumeration item value is a capital letter while the identifier of the related TTCN-3 enumeration value starts with a small letter, the "text '*name*' as capitalized" form shall be used. Otherwise, *freetext* shall contain the value of the related XSD enumeration item.

NOTE 2: The "text '*name*' as uncapitalized", "text '*name*' as lowercased" and "text '*name*' as uppercased" forms are not generated by the current version of the present document but tools are encouraged to support also these encoding instructions for consistency with the "name as … " encoding instruction.

If the first characters of all XSD enumeration items are capital letters, while the names of all related TTCN-3 enumeration values are identical to them except the case of their first characters, the "text all as capitalized" form shall be used.

The encoding instruction designates that the encoder shall use *freetext* or the capitalized name(s) when encoding the TTCN-3 enumeration value(s) and vice versa.

When the text encoding attribute is used with XSD.Boolean types, the decoder shall accept all four possible XSD boolean values and map the received value 1 to the TTCN-3 boolean value **`true`** and the received value 0 to the TTCN-3 boolean value **`false`**. When the text encoding attribute is used on the instances of the XSD.Boolean type, the encoder shall encode the TTCN-3 values according to the encoding attribute (i.e. **`true`** as 1 and **`false`** as 0).

B.3.18 Use number

Syntactical structure(s)

variant """ useNumber """

Applicable to (TTCN-3)

Enumeration types generated for XSD enumeration facets where the enumeration base is integer (see clause 6.1.5, second paragraph).

Description

The encoding instruction designates that the encoder shall use the integer values associated to the TTCN-3 enumeration values to produce the value or the corresponding XML attribute or element (as opposed to the names of the TTCN-3 enumeration values) and the decoder shall map the integer values in the received XML attribute or element to the appropriate TTCN-3 enumeration values.

B.3.19 Use order

Syntactical structure(s)

variant """ useOrder """

Applicable to (TTCN-3)

Record type definition, generated for XSD *complexType*-s with *all* constructor (see clause 7.6.4).

Description

The encoding instruction designates that the encoder shall encode the values of the fields corresponding to the children elements of the *all* constructor according to the order identified by the elements of the **`order`** field. At decoding, the received values of the XML elements shall be placed in their corresponding record fields and a new record of element shall be inserted into the **`order`** field for each XML element processed (the final order of the record of elements shall reflect the order of the XML elements in the encoded XML document).

B.3.20 Whitespace control

Syntactical structure(s)

variant """ whitespace (preserve | replace | collapse) """

Applicable to (TTCN-3)

Types or fields of structured types generated for XSD components with the *whitespace* facet.

Description

The encoding instruction designates that the value of the received XML attribute shall be normalized before decoding as follows (see also clause 3.3.3 of XML 1.1 [5]):

- *preserve*: no normalization shall be done, the value is not changed (this is the behaviour required by XML Schema Part 2 [9] for element content);
- *replace*: all occurrences of HT(9) (horizontal tabulation), LF(10) (line feed) and CR(13) (carriage return) shall be replaced with an SP(32) (space) character;
- *collapse*: after the processing implied by replace, contiguous sequences of SP(32) (space) characters are collapsed to a single SP(32) (space) character, and leading and trailing SP(32) (space) characters are removed.

B.3.21 Untagged elements

Syntactical structure(s)

variant """ untagged """

Applicable to (TTCN-3)

Structured type definitions and structured type fields.

Description

Without this attribute the names of the structured type fields (as possible modified by a **`name as`** and **`namespace`** encoding instructions) or, in case of TTCN-3 type definitions corresponding to global XSD element declarations the name of the TTCN-3 type (as possible modified by a **`name as`** and **`namespace`** encoding instructions) are used as the local part of the start and end tags of XML elements at encoding. If the **`untagged`** encoding instruction is applied to a TTCN-3 type or structured type field, the name of the type or field shall not produce an XML tag when encoding the value of that type or field (in other words, the tag that would be produced without the untagged attribute shall be suppressed during encoding and shall not be expected during decoding). The **`untagged`** encoding instruction shall only have effect on the TTCN-3 language element to which it is directly applied; e.g. if applied to a structured type, the type itself shall not result a starting and end tag in the encoded XML document but the fields of the structured type shall be encoded using starting and end tags (provided no **`untagged`** attribute is applied to the fields).

At decoding no XML starting and end tags shall be present in the encoded XML document. In the specific case, when the **`untagged`** encoding instruction is applied to an optional **`record`** field, which includes merely optional fields, an empty XML *element* shall be decoded to an omitted enframing **`record`** field (see example in clause 7.6.6.6).

Shall not be applied to TTCN-3 components generated for XSD attribute elements (neither global nor local).

For typical use cases for extending or restricting simple content see clauses 7.6.1.1 and 7.6.1.2, for optional sequences see clause 7.6.6.6 and model groups see clause 7.9.

NOTE: Please note, that using the **`untagged`** encoding instruction in other cases than specified in the present document, should result in an undecodable XML document.

B.3.22 Abstract

Syntactical structure(s)

variant """ abstract """

Applicable to (TTCN-3)

Type definitions (generated for global XSD elements and XSD complex types).

Description

This encoding instruction shall have no effect on the encoding process (i.e. it is allowed to send an abstract element or an element with an abstract type to the SUT).

NOTE: Please note that when the "useType" encoding instruction is also appended to the type being used for encoding the element, the *xsi:type* XML attribute will be inserted into the encoded XML element, identifying the name of the abstract XSD type, according to clause B.3.24.

In the decoding process, any of the following cases shall cause a failure of the decoding process:

- the TTCN-3 type corresponding to the XML element to be decoded has both the "element" and "abstract" encoding instructions appended;
- the type of the TTCN-3 field or the field corresponding to the XML element to be decoded has the "abstract" encoding instruction appended and the XML element has no *xsi:type* attribute; or
- if the XML element to be decoded has an *xsi:type* attribute identifying a type to which the "abstract" encoding instruction is appended.

Otherwise the encoding instruction shall have no effect on the decoding process.

B.3.23 Block

Syntactical structure(s)

variant """ block"""

Applicable to (TTCN-3)

Field of the **`union`** type generated for substitutable XSD elements and types.

Description

The encoding instruction shall have no effect on the encoding process.

NOTE: This behaviour is defined to allow sending of intentionally incorrect data to the SUT. Tools may notify the user when the data to be encoded is not valid (a blocked type is used for substitution).

In the decoding process, any of the following cases shall cause a decoding failure:

- the XML element, considering all applied **name** and **namespace** encoding instructions and a possible *xsi:type* XML attribute, would decode to a field of a TTCN-3 **union** type with a "block" encoding instruction;
- the XML element, considering all applied **name** and **namespace** encoding instructions and a possible *xsi:type* XML attribute, would decode to field of a TTCN-3 **union** type without a "block" encoding instruction, but the TTCN-3 type of the field has a "block" encoding instruction.

B.3.24 Use type

Syntactical structure(s)

variant """ useType """"

Applicable to (TTCN-3)

Types, fields of structured types.

Description

The type identification attribute identifies the type of an XML element using the xsi:type attribute from the control namespace (see clause 5.1.5).

In the encoding process **`useType`** instructs the encoder that it shall include the ***xsi:type*** XML attribute into the start tag of the corresponding encoded XML element, with the exception given below. The attribute shall identify the XSD type of the given element, possibly modified in accordance with any final **`name as`** and **`namespace`** encoding instructions. In case of unnamed XSD types the name of the XSD base type shall be used. When **`useType`** is applied to a TTCN-3 **`union`** type, the first alternative of the **`union`** type shall be encoded without an *xsi:type* XML attribute. When **`useType`** is applied to a TTCN-3 **`union`** type supplemented with an **`untagged`** encoding instruction, the **`useType`** encoding instruction shall apply to the alternatives of the **`union`** (i.e. the selected alternative shall be encoded using the *xsi:type* attribute). See examples in clauses 7.5.3 and 8.2. When **`useType`** is applied to a TTCN-3 **`record of`** type with a **`list`** encoding instruction, the *xsi:type* attribute shall be applied to the XML element enclosing the list value. See example in clause 7.5.2.

If a "noType" encoding instruction is applied to the TTCN-3 value to be encoded, the type of which is appended with a useType encoding instruction, the useType instruction shall be ignored.

In the decoding process the presence of the *xsi:type* attribute in an XML element is used in two ways: it shall be used:

a) in the schema validation process of the XML instance to be decoded; and

b) if applied to a TTCN-3 **`union`** type, to select the alternative of the **`union`**, to which the decoded value shall be stowed (see also note in clause 7.5.3). In particular, in the case of type substitution (see clause 8.2), if the XML element to be decoded does not contain an *xsi:type* attribute and it cannot be decoded to the first alternative, the decoding process shall fail (provided no **`useType`** is applied to this field directly). If it is applied to selected alternatives of a **`union`** type but not for the whole type, only these alternatives shall be evaluated taking into account the *xsi:type* attribute.

If used in conjunction with the **useUnion** encoding instruction, the **useType** encoding instruction has no additional effect (the xsi:type attribute is inserted only once). If the selected alternative of the TTCN-3 union type with the **`useType`** encoding instruction is a union type with a final **useUnion** encoding instruction, the type identification attribute shall identify the chosen alternative of the inner union (with the **useUnion** instruction) instead of the alternative of the outer union (with the **useType** encoding instruction).

B.3.25 Process the content of any elements and attributes

Syntactical structure(s)

variant """ processContents (skip | lax | strict) """"

Applicable to (TTCN-3)

XSD.String and record of XSD.String fields of structured types.

Description

The "processContents" encoding instruction controls the validation level of the **content** received at the place of XSD *any* and *anyAttribute* elements at decoding. It has no effect at encoding and does not influence checking the correctness of the namespace of the XML instance being decoded (the namespace shall always satisfy the "anyElement" or "anyAttribute" encoding instruction, see clauses B.3.2 and B.3.3).

If the value of the encoding instruction is "skip", the decoder shall only check if the content is a well-formed XML element or attribute and in case of a defect it shall cause a decoding failure.

If the value of the encoding instruction is "lax", the decoder shall check if the content is well-formed XML element or attribute. If the TTCN-3 definition corresponding to the XML element or attribute being decoded is available for the decoder, the decoder shall also check if the content comply with the TTCN-3 definition. A defect in the well-formedness or in the content validation shall cause a decoding failure. The decoder shall not attempt to retrieve a schema for the element or attribute being decoded from an external source.

If the value of the encoding instruction is "strict", the decoder shall check if the content is well-formed XML element or attribute and, if its content is valid according to the TTCN-3 definition corresponding to the XML element or attribute being decoded. A defect in the well-formedness or in the content validation shall cause a decoding failure. If the corresponding TTCN-3 definition is not available for the decoder, this shall cause a decoding failure. The decoder shall not attempt to retrieve a schema for the element or attribute being decoded from an external source.

B.3.26 Transparent

Syntactical structure(s)

variant """ transparent *name* '*value*' """"

Applicable to (TTCN-3)

Types generated for XSD data types with facet(s) with no direct mapping to TTCN-3.

Description

The "transparent" encoding instruction encapsulates XSD facets that are not directly mapped to TTCN-3 (for directly mapped facets see clause 6, and in particular table 2 of the present document). The *name* part of the instruction shall be the name of the XSD facet and the *value* part of the instruction shall be the value of the facet as defined in XSD (i.e. XSD patterns shall not be converted to TTCN-3 patterns when included into the transparent encoding instruction). In other words, the "transparent" encoding instruction transports the non-mapped XSD facet elements between the XSD specification and the XML codec in a transparent way.

The encoder shall use the content of the "transparent" encoding instruction to generate a correct XML instance for the TTCN-3 value being encoded.

The decoder shall use the "transparent" encoding instruction to validate the received XML document while decoding it.

B.3.27 No Type

Syntactical structure(s)

variant """ noType """

Applicable to (TTCN-3)

Templates, values and fields of templates and values.

Description

The "noType" encoding variant can be applied to any TTCN-3 value or template, where normally an xsi:type attribute would be generated when encoding this element (see clause 5.1.5). This is normally the result of the "useType" or "useUnion" encoding instructions appended to the type of the value or template. This is especially useful for suppressing the type identification attribute for elements derived from simpleType via union. The "noType" encoding instruction takes precedence over the "useType" and "useUnion" encoding instructions.

For decoding purposes, this encoding instruction shall be ignored.

B.3.28 Number of fraction digits

Syntactical structure(s)

variant """ fractionDigits <an integer value>"""

Applicable to (TTCN-3)

Templates, values and fields of templates and values of XSD.Decimal type.

Description

The "fractionDigits" encoding insruction, at encoding constraints the maximum number of digits following the decimal point in the encoded XML element content. TTCN-3 allows using both the decimal point notation and the E-notation for float values (see clause 6.1.0 of ETSI ES 201 873-1 [1]). TTCN-3 values using the E-notation first shall be converted to the dot notation form. In the encoding process, fraction digits excessing the number of fraction digits allowed by the fractionDigits encoding instruction shall simply be truncated.

At decoding, if the received XML value has more fraction digits than allowed by the fractionDigits encoding instruction, it shall cause a decoding failure.

B.3.29 XML header control

Syntactical structure(s)

"noXmlHeader"|"xmlHeader"

Applicable to (TTCN-3)

See description below.

Description

This encoding instruction is not applied to TTCN-3 definitions as a **`variant`** attribute but to the encoding or decoding process, when it is invoked by the **`encvalue`**, **`decvalue`**, **`encvalue_unichar`** and **`decvalue_unichar`** predefined functions (see clause C.5 of ETSI ES 201 873-1 [1]). The encoding instruction shall be passed to the encoding or decoding process in the **`encoding_info`** and d**`ecoding_info`** parameters appropriately. The default value of this encoding instruction is "xmlHeader", i.e. in case of XML encoding this default value overrides the default value "" defined in ETSI ES 201 873-1 [1].

At encoding the instruction controls the presence of the XML prolog, specified in clause 2.8 of the XML 1.1 specification [5], the xsi:schemalocation and namespace specification attributes in the encoded XML value. If its value is "noXmlHeader", neither the XML prolog nor the schemalocation attribute shall be present. Namespace specification (the xmlns attribute) shall be included, if the namespace of the elements to be encoded is specified (by inheriting the "namespace as" encoding instruction from a higher scope unit or it being applied directly; a directly applied instruction takes precedence) (see clause B.3.14 of the present document). If the value is "xmlHeader" - specified directly or by defaulting to it - a complete valid XML documents shall be produced.

At decoding the "noXmlHeader" value instructs the decoder to accept and decode the received XML value as an XML 1.1 document, even without the XML prolog specified in clause 2.8 of the XML 1.1 specification [5], provided all further information (e.g. namespace prefixes and their values) are available for the decoding process. Note that the value "noXmlHeader" has no effect when a complete XML 1.1 document is passed for decoding. The "xmlHeader" value has no effect on the decoding process (i.e. the XML value shall be decoded according to the XML well-formedness and validation rules).

Annex C (informative): Examples

C.0 General

The following examples show how a mapping would look like for example XML Schemas. It is only intended to give an impression of how the different elements have to be mapped and used in TTCN-3.

C.1 Example 1

The XML Schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <!-- This is an embedded example. An element with a sequence body and an attribute.
    The sequence body is formed of elements, two of them are also complexTypes.-->

    <xsd:element name="shiporder">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="orderperson" type="xsd:string"/>
                <xsd:element name="shipto">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="name" type="xsd:string"/>
                            <xsd:element name="address" type="xsd:string"/>
                            <xsd:element name="city" type="xsd:string"/>
                            <xsd:element name="country" type="xsd:string"/>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
                <xsd:element name="item" >
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="title" type="xsd:string"/>
                            <xsd:element name="note" type="xsd:string" minOccurs="0"/>
                            <xsd:element name="quantity" type="xsd:positiveInteger"/>
                            <xsd:element name="price" type="xsd:decimal"/>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="orderid" type="xsd:string" use="required"/>
        </xsd:complexType>
    </xsd:element>

</xsd:schema>
```

Will result in the TTCN-3 module:

```
module NoTargetNamespace {

import from XSD all;


/* This is an embedded example. An element with a sequence body and an attribute.
    The sequence body is formed of elements, two of them are also complexTypes. */

    type record Shiporder
    {
        XSD.String orderid,
        XSD.String orderperson,
        record {
            XSD.String name,
            XSD.String address_,
            XSD.String city,
            XSD.String country
        } shipto,
        record {
```

```
            XSD.String title,
            XSD.String note optional,
            XSD.PositiveInteger quantity,
            XSD.Decimal price
        } item
    }
    with {
        variant "name as uncapitalized";
        variant "element";
        variant (orderid) "attribute";
        variant (shipto.address_) "name as 'address'";
    };

}
with {
    encode "XML";
    variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

The templates in the TTCN-3 module…:

```
module Example1Template {

import from NoTargetNamespace all;

template Shiporder t_Shiporder := {
  orderid := "18920320_17",
  orderperson := "Dr.Watson",
  shipto :=
  {
   name := "Sherlock Holmes",
   address_ := "Baker Street 221B",
   city := "London",
   country := "England"
  },
  item :=
  {
   title := "Memoirs",
   note := omit,
   quantity := 2,
   price := 3.5
  }
}
}//end module
```

can be encoded in XML, for example, as:

```
<?xml version="1.0" encoding="UTF-8"?>
<shiporder orderid='18920320_17'>
    <orderperson>Dr.Watson</orderperson>
    <shipto>
        <name>Sherlock Holmes</name>
        <address>Baker Street 221B</address>
        <city>London</city>
        <country>England</country>
    </shipto>
    <item>
        <title>Memoirs</title>
        <quantity>2</quantity>
        <price>3.500000</price>
    </item>
</shiporder>
```

C.2 Example 2

The XML Schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:tns="http://www.example.org/Part9Example2"
            targetNamespace="http://www.example.org/Part9Example2">

    <xsd:element name="S1" type="tns:S1"/>

    <xsd:simpleType name="S1">
        <xsd:restriction base="xsd:integer">
            <xsd:maxInclusive value="2"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:element name="S2" type="tns:S2"/>

    <xsd:simpleType name="S2">
        <xsd:restriction base="tns:S1">
            <xsd:minInclusive value="-23"/>
            <xsd:maxInclusive value="1"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:element name="S3" type="tns:S3"/>

    <xsd:simpleType name="S3">
        <xsd:restriction base="tns:S2">
            <xsd:minInclusive value="-3"/>
            <xsd:maxExclusive value="1"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:element name="C1" type="tns:C1"/>

    <xsd:complexType name="C1">
        <xsd:simpleContent>
            <xsd:extension base="tns:S3">
                <xsd:attribute name="A1" type="xsd:integer"/>
                <xsd:attribute name="A2" type="xsd:float"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>

</xsd:schema>
```

Will result in the TTCN-3 module:

```
module http_www_example_org_Part9Example2 {

    import from XSD all;

    type S1_1 S1
    with {
        variant "element";
    };

    type XSD.Integer S1_1 (-infinity .. 2)
    with {
        variant "name as 'S1'";
    };

    type S2_1 S2
    with {
        variant "element";
    };

    type S1_1 S2_1 (-23 .. 1)
    with {
    variant "name as 'S2'";
    };

    type S3_1 S3
    with {
```

```
        variant "element";
    };

    type S2_1 S3_1 (-3 .. 1)
    with {
        variant "name as 'S3'";
    };

    type C1_1 C1
    with {
    variant "element";
    };

    type record C1_1
    {
        XSD.Integer a1 optional,
        XSD.Float a2 optional,
        S3_1 base
    }
    with {
        variant "name as 'C1'";
        variant (a1, a2) "name as capitalized";
        variant (a1, a2) "attribute";
        variant (base) "untagged";
    };

}
with {
    encode "XML";
    variant "namespace as 'http://www.example.org/Part9Example2' prefix 'tns'";
    variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

The template in the TTCN-3 module…:

```
module Example2Templates {

    import from http_www_example_org_Part9Example2 all;

    template C1 t_C1:= {
        a1 :=1,
        a2 :=2.0,
        base :=-1
    }
}
```

can be encoded in XML, for example, as:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:C1 xmlns:tns='http://www.example.org/Part9Example2' A1='1' A2='2.000000'>-1</tns:C1>
```

C.3 Example 3

The XML Schemas:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:tns="http://www.example.org/Part9Example3"
               targetNamespace="http://www.example.org/Part9Example3">

    <xsd:element name="C1" type="tns:C1"/>

    <xsd:complexType name="C1">
        <xsd:simpleContent>
            <xsd:extension base="xsd:integer">
                <xsd:attribute name="A1" type="xsd:integer"/>
                <xsd:attribute name="A2" type="xsd:integer"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>

    <xsd:element name="C2" type="tns:C2"/>

    <xsd:complexType name="C2">
```

```
        <xsd:simpleContent>
            <xsd:restriction base="tns:C1">
                <xsd:minInclusive value="23"/>
                <xsd:maxInclusive value="26"/>
                <xsd:attribute name="A1" type="xsd:byte" use="required"/>
                <xsd:attribute name="A2" type="xsd:negativeInteger"/>
            </xsd:restriction>
        </xsd:simpleContent>
    </xsd:complexType>

    <xsd:element name="C3" type="tns:C3"/>

    <xsd:complexType name="C3">
        <xsd:simpleContent>
            <xsd:restriction base="tns:C2">
                <xsd:minInclusive value="25"/>
                <xsd:maxInclusive value="26"/>
            </xsd:restriction>
        </xsd:simpleContent>
    </xsd:complexType>

</xsd:schema>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:tns="http://www.example.org/Part9Example3"
               xmlns:ns2="http://www.example.org/Part9Example2"
               targetNamespace="http://www.example.org/Part9Example3">

    <xsd:include schemaLocation="./Part9Example3a.xsd"/>
    <xsd:import schemaLocation="./Part9Example2.xsd"
                namespace="http://www.example.org/Part9Example2"/>

    <xsd:element name="newC1" type="tns:newC1"/>

    <xsd:complexType name="newC1">
        <xsd:complexContent>
            <xsd:extension base="tns:C1"/>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:element name="newS1" type="tns:newS1"/>

    <xsd:simpleType name="newS1">
        <xsd:restriction base="ns2:S1"/>
    </xsd:simpleType>

</xsd:schema>
```

Will result in the TTCN-3 module:

NOTE: Please note that the two schemas above have the same target namespace, therefore they will result in one generated TTCN-3 module.

```
module http_www_example_org_Part9Example3 {

    import from XSD all;
    import from http_www_example_org_Part9Example2 all;

    type C1_1 C1
    with {
        variant "element";
    };

    type record C1_1
    {
        XSD.Integer a1 optional,
        XSD.Integer a2 optional,
        XSD.Integer base
    }
    with {
        variant "name as 'C1'";
        variant (a1, a2) "name as capitalized";
        variant (a1, a2) "attribute";
        variant (base) "untagged";
    };
```

```
    type C2_1 C2
    with {
        variant "element";
    };

    type record C2_1
    {
        XSD.Byte a1,
        XSD.NegativeInteger a2 optional,
        XSD.Integer base (23 .. 26)
    }
    with {
        variant "name as 'C2'";
        variant (a1, a2) "name as capitalized";
        variant (a1, a2) "attribute";
        variant (base) "untagged";
    };

    type C3_1 C3
    with {
        variant "element";
    };

    type record C3_1
    {
        XSD.Byte a1,
        XSD.NegativeInteger a2 optional,
        XSD.Integer base (25 .. 26)
    }
    with {
        variant "name as 'C3'";
        variant (a1, a2) "name as capitalized";
        variant (a1, a2) "attribute";
        variant (base) "untagged";
    };

    type NewC1_1 NewC1
    with {
        variant "name as uncapitalized";
        variant "element";
    };

    type record NewC1_1
    {
    XSD.Integer a1 optional,
    XSD.Integer a2 optional,
    XSD.Integer base
    }
    with {
        variant "name as 'newC1'";
        variant (a1, a2) "name as capitalized";
        variant (a1, a2) "attribute";
        variant (base) "untagged";
    };

    type NewS1_1 NewS1
    with {
        variant "name as uncapitalized";
        variant "element";
    };

    type S1_1 NewS1_1
    with {
        variant "name as 'newS1'";
    };

}
with {
    encode "XML";
    variant "namespace as 'http://www.example.org/Part9Example3' prefix 'tns'";
    variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
}
```

The templates in the TTCN-3 module…:

```
module Example3Templates {

import from http_www_example_org_Part9Example3 all;

    template C1 t_C1:= {
        a1 := 1,
        a2 := 2,
        base := -99
    }


    template C2 t_C2:= {
        a1 := 10,
        a2 := -20,
        base := 24
    }

    template C3 t_C3:= {
        a1 := 127,
        a2 := -200,
        base := 25
    }

    template NewC1 t_NewC1:= {
        a1 := 100,
        a2 := 200,
        base := -999
    }

    template NewS1 t_NewS1 := 1

}
```

can be encoded in XML, for example, as:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:C1 xmlns:tns='http://www.example.org/Part9Example3' A1='1' A2='2'>-99</tns:C1>


<?xml version="1.0" encoding="UTF-8"?>
<tns:C2 xmlns:tns='http://www.example.org/Part9Example3' A1='10' A2='-20'>24</tns:C2>


<?xml version="1.0" encoding="UTF-8"?>
<tns:C3 xmlns:tns='http://www.example.org/Part9Example3' A1='127' A2='-200'>25</tns:C3>


<?xml version="1.0" encoding="UTF-8"?>
<tns:newC1 xmlns:tns='http://www.example.org/Part9Example3' A1='100' A2='200'>-999</tns:newC1>


<?xml version="1.0" encoding="UTF-8"?>
<tns:newS1 xmlns:tns='http://www.example.org/Part9Example3'>1</tns:newS1>
```

Annex D (informative): Deprecated features

D.1 Using the anyElement encoding instruction to record of fields

The TTCN-3 core language, ETSI ES 201 873-1 [1], up to and including V3.4.1, did not allow referencing the type replicated in a TTCN-3 record of or set of type definition. As a consequence, when the *any* XSD element have had a maxOccurs attribute with the value more then 1 (including "unbounded"), and is converted to a TTCN-3 **`record of`** `XSD.String` field, the `anyElement` encoding instruction could not be attached to the XSD.String type, as in all other cases, but have had to be attached to the **`record of`**. As the above limitation was removed in the core language, using the `anyElement` encoding instruction with other types than the XSD.String, resulted from the conversion of an XSD *any* element is deprecated. TTCN-3 tools, however, are encouraged to accept both syntaxes in TTCN-3 modules further on, but, when converting XSD Schemas to TTCN-3, generate only the syntax according to the present document.

EXAMPLE 1: The outdated syntax:

```
<xsd:complexType name="e46b">
    <xsd:sequence>
        <xsd:any minOccurs="0" maxOccurs="unbounded" namespace="##local"/>
    </xsd:sequence>
</xsd:complexType>

//Was mapped to the following TTCN-3 code and encoding extensions according to
//elder versions of the present document:
type record E46b {
    record of XSD.String elem_list
}
with {
    variant "name as uncapitalized";
    variant(elem_list) "anyElement except unqualified"
}
```

EXAMPLE 2: The present syntax:

```
<xsd:complexType name="e46b">
    <xsd:sequence>
        <xsd:any minOccurs="0" maxOccurs="unbounded" namespace="##local"/>
    </xsd:sequence>
</xsd:complexType>

//Is mapped to the following TTCN-3 code and encoding extensions:
type record E46b {
    record of XSD.String elem_list
}
with {
    variant "name as uncapitalized";
    variant(elem_list[-]) "anyElement except unqualified"
//                    ^ ^  pls. note the dash syntax here
}
```

D.2 Using the XML language identifier string

When importing from an XSD Schema, previous versions of the present document (up to V4.3.1) required to use the following language identifier strings:

- "XML" or "XML1.0" for W3C® XML 1.0; and
- "XML1.1" for W3C® XML 1.1.

These strings are deprecated and have been replaced by another string (see clause 5) and may be fully removed in a future edition of the present document.

NOTE: Please note, that the encoding attribute values associated with the XSD to TTCN-3 language mapping specified in the present document remain unchanged (see clause B.2).

D.3 Id

This earlier mapping of the *id* XSD attribute to a TTCN-3 type has been deprecated. It has no bearing on testing using XML documents.

Annex E (informative): Bibliography

- World Wide Web Consortium W3C Recommendation (2005): "xml:id Version 1.0".

NOTE: Available at http://www.w3.org/TR/xml-id/.

- ISO/IEC 646: "Information technology - ISO 7-bit coded character set for information interchange".
- ETSI ES 202 782: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: TTCN-3 Performance and Real Time Testing".
- ETSI ES 202 784: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Advanced Parameterization".
- ETSI ES 202 785: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Behaviour Types".
- ETSI ES 202 786: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Support of interfaces with continuous signals".
- ETSI ES 202 789: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Extended TRI".

History

Document history		
V3.3.1	July 2008	Publication
V4.1.1	June 2009	Publication
V4.2.1	July 2010	Publication
V4.3.1	June 2011	Publication
V4.4.1	April 2012	Publication
V4.5.1	April 2013	Publication
V4.6.1	June 2015	Publication
V4.7.1	July 2016	Publication
V4.8.1	May 2017	Publication
V4.9.1	March 2018	Membership Approval Procedure MV 20180525: 2018-03-26 to 2018-05-25