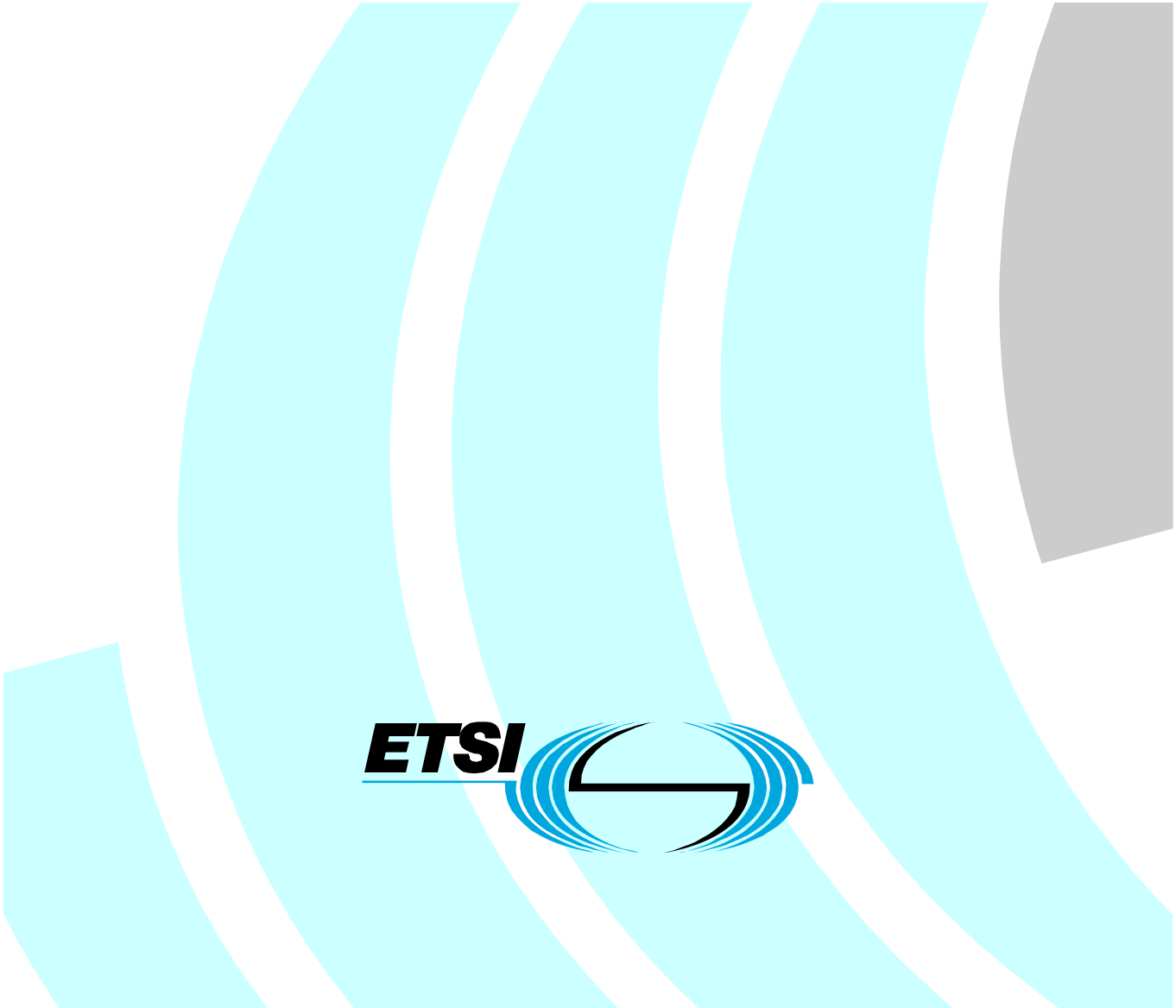
ETSI ES 201 873-9 V4.2.1 (2010-07)

ETSI Standard

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-00111-9 T3 ed421 XML

Keywords
MTS, testing, TTCN, 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

Important notice

Individual copies of the present document can be downloaded from:
http://www.etsi.org

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

Users of the present document should be aware that the document may be subject to revision or change of status. Information on the current status of this and other ETSI documents is available at
http://portal.etsi.org/tb/status/status.asp

If you find errors in the present document, please send your comment to one of the following services:
http://portal.etsi.org/chaircor/ETSI_support.asp

Copyright Notification

Contents

Intellectual Property Rights 7
Foreword 7
1 Scope 8
2 References 8
2.1 Normative references 8
2.2 Informative references 9
3 Definitions and abbreviations 9
3.1 Definitions 9
3.2 Abbreviations 9
4 Introduction 10
4.1 Conformance and compatibility 10
5 Mapping XML Schemas 10
5.1 Namespaces and document references 11
5.2 Name conversion 14
5.2.1 General 14
5.2.2 Name conversion rules 15
5.2.3 Order of the mapping 19
5.3 Unsupported features 20
6 Built-in data types 20
6.1 Mapping of facets 20
6.1.1 Length 21
6.1.2 MinLength 22
6.1.3 MaxLength 22
6.1.4 Pattern 22
6.1.5 Enumeration 24
6.1.6 WhiteSpace 26
6.1.7 MinInclusive 26
6.1.8 MaxInclusive 28
6.1.9 MinExclusive 29
6.1.10 MaxExclusive 30
6.1.11 Total digits 31
6.2 String types 32
6.2.1 String 32
6.2.2 Normalized string 32
6.2.3 Token 32
6.2.4 Name 33
6.2.5 NMTOKEN 33
6.2.6 NCName 33
6.2.7 ID 33
6.2.8 IDREF 33
6.2.9 ENTITY 33
6.2.10 Hexadecimal binary 33
6.2.11 Base 64 binary 34
6.2.12 Any URI 34
6.2.13 Language 34
6.2.14 NOTATION 34
6.3 Integer types 34
6.3.1 Integer 34
6.3.2 Positive integer 35
6.3.3 Non-positive integer 35
6.3.4 Negative integer 35
6.3.5 Non-negative integer 35
6.3.6 Long 35

6.3.7 Unsigned long 35
6.3.8 Int 35
6.3.9 Unsigned int 36
6.3.10 Short 36
6.3.11 Unsigned Short 36
6.3.12 Byte 36
6.3.13 Unsigned byte 36
6.4 Float types 36
6.4.1 Decimal 36
6.4.2 Float 37
6.4.3 Double 37
6.5 Time types 37
6.5.1 Duration 37
6.5.2 Date and time 38
6.5.3 Time 38
6.5.4 Date 38
6.5.5 Gregorian year and month 38
6.5.6 Gregorian year 38
6.5.7 Gregorian month and day 38
6.5.8 Gregorian day 39
6.5.9 Gregorian month 39
6.6 Sequence types 39
6.6.1 NMTOKENS 39
6.6.2 IDREFS 39
6.6.3 ENTITIES 39
6.6.4 QName 39
6.7 Boolean type 40
6.8 AnyType and anySimpleType types 40

7 Mapping XSD components 41
7.1 Attributes of XSD component declarations 41
7.1.1 Id 42
7.1.2 Ref 42
7.1.3 Name 42
7.1.4 MinOccurs and maxOccurs 42
7.1.5 Default and Fixed 44
7.1.6 Form 45
7.1.7 Type 46
7.1.8 Mixed 46
7.1.9 Abstract 46
7.1.10 Block and final 46
7.1.11 Nillable 46
7.1.12 Use 48
7.1.13 Substitution group 48
7.2 Schema component 48
7.3 Element component 48
7.4 Attribute and attribute group definitions 50
7.4.1 Attribute element definitions 50
7.4.2 Attribute group definitions 50
7.5 SimpleType components 50
7.5.1 Derivation by restriction 50
7.5.2 Derivation by list 51
7.5.3 Derivation by union 52
7.6 ComplexType components 54
7.6.1 ComplexType containing simple content 54
7.6.1.1 Extending simple content 54
7.6.1.2 Restricting simple content 55
7.6.2 ComplexType containing complex content 55
7.6.2.1 Complex content derived by extension 55
7.6.2.2 Complex content derived by restriction 60
7.6.3 Referencing group components 61
7.6.4 All content 64

7.6.5 Choice content 65
7.6.5.1 Choice with nested elements 66
7.6.5.2 Choice with nested group 66
7.6.5.3 Choice with nested choice 67
7.6.5.4 Choice with nested sequence 67
7.6.5.5 Choice with nested any 68
7.6.6 Sequence content 69
7.6.6.1 Sequence with nested element content 69
7.6.6.2 Sequence with nested group content 69
7.6.6.3 Sequence with nested choice content 70
7.6.6.4 Sequence with nested sequence content 70
7.6.6.5 Sequence with nested any content 71
7.6.6.6 Effect of the *minOccurs* and *maxOccurs* attributes on the mapping 71
7.6.7 Attribute definitions, attribute and attributeGroup references 73
7.6.8 Mixed content 75
7.7 Any and anyAttribute 78
7.8 Annotation 80
7.9 Group components 80
7.10 Identity-constraint definition schema components 81

Annex A (normative): TTCN-3 module XSD 82

Annex B (normative): Encoding instructions 86

B.1 General 86

B.2 The XML encode attribute 86

B.3 Encoding instructions 87
B.3.1 XSD data type identification 87
B.3.2 Any element 87
B.3.3 Any attributes 88
B.3.4 Attribute 88
B.3.5 AttributeFormQualified 89
B.3.6 Control Namespace identification 89
B.3.7 Default for empty 89
B.3.8 Element 89
B.3.9 ElementFormQualified 90
B.3.10 Embed values 90
B.3.11 Form 91
B.3.12 List 91
B.3.13 Name 91
B.3.14 Namespace identification 92
B.3.15 Nillable elements 92
B.3.16 Use Union 93
B.3.17 Text 93
B.3.18 Use number 94
B.3.19 Use order 94
B.3.20 Whitespace control 94
B.3.21 Untagged elements 95

Annex C (informative): Examples 96

C.1 Example 1 96

C.2 Example 2 97

C.3 Example 3 99

C.4 Example 4 100

Annex D (informative): Deprecated features 102
D.1 Using the anyElement encoding instruction to record of fields 102
History 103

Intellectual Property Rights

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

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

Foreword

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

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

1 Scope

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

2 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 http://docbox.etsi.org/Reference.

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

2.1 Normative references

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

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

[2] ETSI ES 201 873-7: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 7: Using ASN.1 with TTCN-3".

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

[4] ITU-T Recommendation 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

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

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".

3 Definitions and abbreviations

3.1 Definitions

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

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: contains 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: represented by a 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).

3.2 Abbreviations

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

ASN.1	Abstract Syntax Notation One
DTD	Document Type Description
SOAP	Simple Object Access Protocol
TTCN-3	Testing and Test Control Notation version 3
URI	Uniform Resource Identifier
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], [8] 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 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 part 7 [2] of this multi-part deliverable. 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.

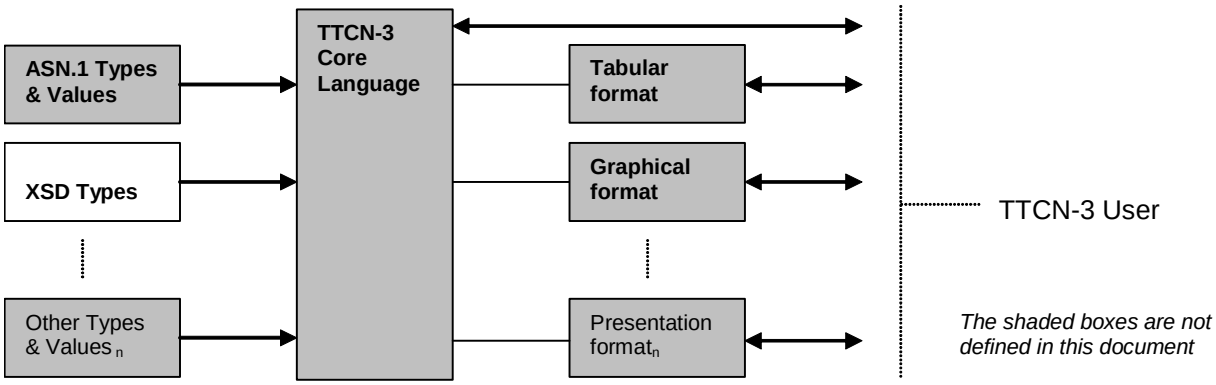


Figure 1: User's view of the core language and the various presentation formats

For compatibility reasons, the TTCN-3 code obtained from the XML Schema using the present document for an explicit mapping shall be the same as the TTCN-3 code obtained from first converting the XML Schema using ITU-T Recommendation X.694 [4] into ASN.1 [3] then converting the resulting ASN.1 into TTCN-3 code using ES 201 873-7 [2]. Moreover, the XML document produced by the TTCN-3 code with encoding extensions obtained from the XML Schema based on the present document shall be the same as the XML document produced by the ASN.1 with E-XER encoding based on ITU-T Recommendation X.694 [4] conversion of the same XML Schema.

4.1 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 ES 201 873-1 [1].

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

ETSI ES 201 873-1 [1], version 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.

5 Mapping XML Schemas

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 not accessible by the user.

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. Built-in data types, described in detail in clause 6, in case of an implicit conversion are internal to the tool and can be referenced directly by the user, while in case of an explicit conversion, the user shall have to import the XSD.ttcn module (see annex A) in addition to the TTCN-3 modules resulted from the conversion. When importing from an XSD Schema, the following language identifier strings shall be used:

- "XML" or "XML1.0" for W3C XML 1.0; and
- "XML1.1" for W3C XML 1.1.

All XSD definitions are **`public`** by default (see clause 8.2.3 of 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 two 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.

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

A single XSD Schema shall be translated to one or more TTCN-3 modules, corresponding to schema components that have the same target namespace, including no target namespace, i.e. one TTCN-3 module shall be generated for each target namespace (including absence of the target namespace) of the XML Schema. The names of the generated TTCN-3 modules 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.

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 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 2: The above statement means that XSD components using imported XSD references shall be handled as complete, i.e. it is not needed to 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 3: 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 ES 201 873-1 [1]), or an import all statement contains an exception list (see clause 8.2.3.5 of 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 4: 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 like defined for TTCN-3, see clause 8.2.3.7 of ES 201 873-1 [1]).

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*.

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 5: If different *schema* elements 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 1: Schemas with the same namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="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 -->
:
</schema>

<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="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 -->
:
</schema>

//Will result the TTCN-3 module
module 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'
}
```

EXAMPLE 2: A schema with a target namespace is including a schema without a target namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:ns1="http://www.example.org"
            targetNamespace="http://www.example.org">
    <!-- the including schema -->
    <include schemaLocation="included.xsd"/>
:
</schema>

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

//Will result the TTCN-3 modules (please note, the content of the modules may come
// from more than one schemas.
module 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"
}
```

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).

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

EXAMPLE 3: Mapping of schema attributes:

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

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

```
        <attribute name="attrib" type="integer"/>
    </complexType></schema>
</schema>

//Will result in the TTCN-3 modules (please note, that the content of the modules may come
//from more than one schemas.
module 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"
}
```

The control namespace is the namespace of the type identification attributes (see clause 7.5.3) and to be used for schema instances (e.g. in the special XML attribute value "xsi:nil", see mapping of the *nillable* XSD attribute in clause 7.1.11). It shall be specified globally, with an encoding instruction attached to the TTCN-3 module.

NOTE 6: For the special XML attribute value "xsi:nil", see mapping of the *nillable* XSD attribute in clause 7.1.11).

EXAMPLE 4: 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 must guarantee TTCN-3 consistency, but, in order to support migration of conversion results from code generated with tools based on ITU-T Recommendation X.694 [4], it must also generate identifiers compatible with that standard. It must 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;

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;

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

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

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

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: 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.

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) and "-" (HYPEN-MINUS) shall all be replaced by a "_" (LOW LINE);

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;

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).

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

j) 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 one of the reserved words specified in clause 11.27 of ITU-T Recommendation 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], 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.

k) 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, 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, 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 ES 201 873-1 [1]) or names of predefined functions (see clause 16.1.2 of ES 201 873-1 [1]) after applying the postfix to clashing field names, shall be suffixed by a single "_" (LOW LINE) character.

l) If the name being generated is the identifier of an enumeration item (see clause 6.2.4 of 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, 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 ES 201 873-1 [1]) or names of predefined functions (see clause 16.1.2 of ES 201 873-1 [1]) after applying the postfix to clashing enumeration names, shall be suffixed by a single "_" (LOW LINE) character.

EXAMPLE 1: Conversion of an XML Schema composed of two schema elements with identical target namespaces:

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

```
            <attribute name="Foobar" type="integer"/>
    </complexType>
</schema>

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

//Will be translated to:
module 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 '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 2: 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>

//is mapped to the TTCN-3 type assignment:
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 3: 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 1: 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 target namespace, 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 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 *abstract* attribute is not supported.

e) Translation of the *block* attribute is not supported.

f) Translation of the *final* attribute is not supported.

g) Translation of the *substitutionGroup* attribute and element substitution is not supported.

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

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

6 Built-in data types

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:

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

Table 2 summarizes the facets for the built-in types that are supported in TTCN-3. Some of them may be supported in XML Schema but have no counterpart in TTCN-3 and therefore no mark 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 must 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:

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

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

//Mapped to TTCN-3:
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:

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

//Is translated to:
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:

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

//Is mapped to:
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 must 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 ES 201 873-1 [1].

EXAMPLE:

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

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 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, 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.

Any other enumeration facet shall be mapped to value list subtyping, if this is allowed by 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 ES 201 873-1 [1], the enumeration facet shall be ignored.

NOTE: 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"
}
```

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 uncapitalized";
    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 uncapitalized"
}
```

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:minExclusive 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 uncapitalized"
}
```

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:

```
<simpleType name="e8">
    <restriction base="string">
        <whiteSpace value="replace"/>
    </restriction>
</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);

h) 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]).

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

NOTE: Note, that 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:

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

//Is mapped to:
type XSD.Integer E9a (-5 .. infinity)
with {
    variant "name as uncapitalized"
}
```

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

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

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

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

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

//Is mapped to:
type XSD.Float E9c (-infinity .. infinity)
with {
    variant "name as uncapitalized";
}

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

//Is mapped to:
type XSD.Float E9d ( infinity )
with {
    variant "name as uncapitalized";
}

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

//Is mapped to:
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);

j) 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]).

k) 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 **`-3.40282346638528859811704183484 52E38`** ((2^24-1)*2^104) and of the XSD *double* type is **`-1.82687704666362844493051000437 86E47`** ((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:

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

//Is mapped to:
type XSD.PositiveInteger E10a (1 .. 100)
with {
    variant "name as uncapitalized"
}
```

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

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

//Is mapped to:
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:

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

```
//Is mapped to:
type XSD.Float E10c (-infinity .. infinity)
with {
    variant "name as uncapitalized";
}

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

//Is mapped to:
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.

l) 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;

m) 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]).

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

NOTE 2: Note, that the upper bound of the value space of the XSD *float* type is **`3.40282346638528859811704183484 52E38`** ((2^24-1)*2^104) and of the XSD *double* type is **`1.82687704666362844493051000437 86E47`** ((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:

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

//Is mapped to TTCN-3 as:
type XSD.Integer E11a (!-5 .. infinity)
with {
    variant "name as uncapitalized"
}
```

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

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

//Is mapped to TTCN-3 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"
}

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

//Is mapped to TTCN-3 as:
type XSD.Float E11c (!-6.0 .. -5.0)
with {
    variant "name as uncapitalized"
}

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

o) 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.

p) In case e) 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]).

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

NOTE 2: Note, that the lower bound of the value space of the XSD *float* type is **`-3.40282346638528859811704183484 52E38`** ((2^24-1)*2^104) and of the XSD *double* type is **`-1.82687704666362844493051000437 86E47`** ((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*:

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

// Is mapped in TTCN-3 to:
type XSD.PositiveInteger E12a (1 .. !100)
with {
    variant "name as uncapitalized"
}
```

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

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

// Is mapped in TTCN-3 to:
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"
}

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

// Is mapped in TTCN-3 to:
type XSD.Float E12c (-5.0 .. ! -4.0)
with {
    variant "name as uncapitalized"
}

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

// No corresponding TTCN-3 type is produced.

6.1.11 Total digits

This facet defines the total number of digits a numeric value is allowed to have. 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.

EXAMPLE:

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

// Is translated to:
type XSD.NegativeInteger E13 (-999 .. -1)
with {
    variant "name as uncapitalized"
}
```

6.2 String types

XSD string types shall generally be converted to TTCN-3 as subtypes of *universal charstring* or *octetstring*. For an overview of the allowed facets please refer to table 2. 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 D.2.2.0 of 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(0,0,0,12)..char(0,0,0,12),
  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:

```
<simpleType name="E14">
<restriction base="base64Binary"/>
</simpleType>

//Is  translated as:

type XSD.Base64Binary E14;

// and the template:

template E14 MyBase64BinaryTemplate := '546974616E52756C6573'O

// Is encoded as:

<E14>VGl0YW5SdWxlcw==\r\n</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

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. The following clauses 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 D.2.1.3 of 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 D.2.1.3 of 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 D.2.1.2 of 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 D.2.1.2 of 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 D.2.1.1 of 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 D.2.1.1 of 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 D.2.1.0 of 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 D.2.1.0 of ES 201 873-1 [1]:

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

6.4 Float types

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. 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
with {
    variant "XSD:decimal"
}
```

6.4.2 Float

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

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

6.5 Time types

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(0(0[1-9]|[1-9][0-9])|[1-9][0-9][0-9])|[1-9][0-9][0-9][0-9])",
  yearExpansion := "(-([1-9][0-9]#(0,))#(,1))#(,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))"
```

NOTE 1: The patterns below implement the syntactical restrictions of ISO 8601 [i.2] and XSD (e.g. year 0000, 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: 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.

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

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. 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:

```
type record E14a
{
    QName name,
    integer refId
}

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 as an XML compatible restricted subtype of the universal charstring.

EXAMPLE:

```
type XSD.XMLCompatibleString AnySimpleType
with {
    variant "XSD:anySimpleType"
}

//while anyType is translated into XML content opaque to the codec:

type record AnyType {
    record of XSD.String attr,
    record of XSD.String elem_list
}
```

```
with {
    variant "XSD:anyType";
    variant(attr) "anyAttributes";
    variant(elem_list) "anyElement";
}
```

See also clause 7.7.

7 Mapping XSD components

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

Tables 5 and 6 contain an overview about the major attributes that shall be encountered during mapping is the tables are incomplete, the special attributes that are only used by a single XSD component 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
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)						

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 attribute *id* enables a unique identification of an XSD component. They shall be mapped to TTCN-3 as simple type references, e.g. any component mapping to a type with name **typeName** and an attribute id="**ID**" shall result in an additional TTCN-3 type declaration:

```
type <Typename> ID;
```

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.

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.

When 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 **`record of`** shall be unrestricted if *minOccurs* equals to "0" and *maxOccurs* equals to "unbounded" and shall be length restricted otherwise (where *maxOccurs*="unbounded" shall be translated to the upper bound **`infinity`**). The initial name of the field shall be 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 *'<initial name>'*" encoding instruction shall be attached to the inner type, where *<initial name>* is the name resulted from applying clause 5.2.2 to the name of the XSD component being translated.

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: Please note, that 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	TTCN-3	mapping
		TTCN-3 construct	preserved field name postfix
0	0		
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
<*x*> ≠ 0	<*y*> ≠ 1	**record length** (<*x*>..<*y*>) **of** *<initial type>*	_list
<*x*> ≥ 1	unbounded	**record length** (<*x*>..**infinity**) **of** *<initial type>*	_list

EXAMPLE 1: Mapping of an optional *element*:

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

// Is translated to an optional field as:

type record E15 {
    XSD.Integer foo optional,
    XSD.Float  bar
}
with {
    variant "name as uncapitalized"
}
```

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

```
<!-- The unrestricted case: -->
<complexType name="e15b">
    <sequence>
        <element name="foo" type="integer" minOccurs="0" maxOccurs="unbounded"/>
        <element name="bar" type="float"/>
    </sequence>
</complexType>

// Is translated to TTCN-3 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: -->
<complexType name="e15c">
    <sequence>
        <element name="foo" type="integer" minOccurs="5" maxOccurs="10"/>
        <element name="bar" type="float"/>
    <sequence>
</complexType>

// Is translated to TTCN-3 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 we have: -->
<group name="foobarGroup">
  <sequence>
    <element name="foo" type="string"/>
    <element name="bar" type="string"/>
  </sequence>
</group>

<!-- The optional case: -->
<complexType name="e15d">
    <group ref="ns:foobarGroup" minOccurs="0"/>
</complexType>

// Is translated to TTCN-3 as:
type record FoobarGroup {
    XSD.String foo,
    XSD.String bar
}
with {
    variant "untagged"
    //pls. 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:

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

// Is translated to TTCN-3 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.
}
```

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:

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

// Is be translated to:

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 must 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

Mapping of the XSD *abstract* attribute is not supported by the present document.

7.1.10 Block and final

Mapping of the XSD *block* and *final* attributes are not supported by the present document.

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 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 of the element if the value of the *nillable* attribute would be "false". The **`record`** type or field shall be appended with the "useNil" encoding instruction.

EXAMPLE 1: Mapping of *nillable* elements:

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

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


//Are translated to TTCN-3 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(bar) "useNil"
}

// Which allows e.g. the following encoding:
template E16a t_E16a :=
{
    foo:=3,
    bar:= { content := omit }
}

<?xml version="1.0" encoding="UTF-8"?>
    <e16a>
        <foo>3</foo>
        <bar xsi:nil="true"/>
    </e16a>
```

EXAMPLE 2: Joint use of the *nillable*, *minOccurs* and *maxOccurs* attributes:

```
<element name="SeqNillable" nillable="true">
    <complexType>
        <sequence>
            <element name="forename" type="string" nillable="true"/>
            <element name="surname" type="string" minOccurs="0" nillable="true"/>
            <element name="bornPlace"   type="string" minOccurs="0" maxOccurs="unbounded"
                                                nillable="true"/>
            <element ref="ns:remarkNillable"/>
        </sequence>
    </complexType>
</element>

//Is translated to TTCN-3 as:
type record SeqNillable {
    record {
        record {
            XSD.String content optional
        } forename,
        record {
            XSD.String content optional
        } surname optional,
        record of record {
            XSD.String content optional
        } bornPlace_list,
        record {
            XSD.String content optional
        } remarkNillable
    } content optional
}
```

```
with {
    variant "element";
    variant "useNil";
    variant(content.bornPlace_list) "name as'bornPlace'";
    variant(content.forename, content.surname, content.bornPlace_list, content.remarkNillable)
           "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 (for an example see clause 7.6.2.2).

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 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

Mapping of the XSD *substitutionGroup* attribute is not supported by the present document, i.e. this attribute shall be ignored when the element is translated to TTCN-3.

NOTE: TTCN-3 supports this XSD feature via type compatibility, unless the substitution element is derived by extending the head element.

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:

- 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.
- In case of referenced XSD user types, the TTCN-3 type generated for the referenced XSD type.
- In case the child of the *element* is a locally defined *complexType*, it shall be a TTCN-3 **record**.

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:

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

// is translated to:
type typename E16a
with {
    variant "element";
    ariant "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 resulted by mapping the type of the XSD *element* 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.3.1.4), 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):

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


//Is translated into:
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 in the same way as elements (see clause 7.3), except that they shall additionally be appended with the “attribute” TTCN-3 encoding instruction (instead of the "element" instruction attached to TTCN-3 types resulted by mapping of *element*s).

EXAMPLE: Mapping of a globally defined attribute:

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

// is mapped to:
type typename E17
with {
    variant "attribute";
    variant "name as uncapitalized "
}
```

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

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

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.

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: Consider the following example restricting an anonymous *simpleType* using a pattern facet (the bold part marks the inner **simpleType**):

```
<simpleType name="e18">
    <restriction base="string"/>
        <pattern value="(aUser|anotherUser)@(i|I)nstitute"/>
    </restriction>
</simpleType>

// This will generate a mapping for the inner type and a restriction thereof:
type XSD.String E18 (pattern "(aUser|anotherUser)@(i|I)nstitute")
with {
    variant "name as uncapitalized "
}
```

7.5.2 Derivation by list

XSD list components shall be mapped to the TTCN-3 *record of* type. In their simplest form lists shall be mapped by directly using the *listItem* attribute as the resulting type.

EXAMPLE 1:

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

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

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.

EXAMPLE 2: Consider this example:

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

// Will map to:
type record length(3) of XSD.Float E20
with {
    variant "list";
    variant "name as uncapitalized"
}

//For instance the template:

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

// will be encoded as:

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

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).

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).

NOTE 1: XSD requires (see XML Schema Part 2: Datatypes [9], clause 2.5.1.3) that an element or attribute value is validated against the member types in the order in which they appear in the definition until a match is found. 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.

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.

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

```
<!-- Please compare the mapping of the two definitions below -->
<xsd:simpleType name="e21memberlist">
    <xsd:union memberTypes="xsd:integer xsd:boolean"/>
</xsd:simpleType>

<simpleType name="e21unnamed">
    <union>
        <simpleType>
            <restriction base="string"/>
        </simpleType>
        <simpleType>
            <restriction base="float"/>
        </simpleType>
    </union>
</simpleType>

// Results in the following mappings:
type union E21memberlist {
    XSD.Integer integer_,
    XSD.Boolean boolean_
}
with {
    variant "name as uncapitalized";
    variant "useUnion"
}

type union E21unnamed {
    XSD.String alt_,
    XSD.Float alt_1
}
with {
    variant "name as uncapitalized";
    variant "useUnion"
    variant(alt_, alt_1) "name as ''"
}

// For instance, the below structure:
type record E21a {
    E21unnamed e21unnamed,
    XSD.String foo
}
with {
    variant "name as uncapitalized";
    variant "element"
}

template E21a t_E21a:={
    e21unnamed := { alt_ := "ding" },
    foo:="foostring"
}
```

```
// will result in the following encoding:
<?xml version="1.0" encoding="UTF-8"?>
<e21a xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'>
<e21unnamed xsi:type="string">ding</e21unnamed>
<foo>foostring</foo>
</e21a>
```

EXAMPLE 2: 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:minInclusive value="2003-01-01"/>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:union>
</xsd:simpleType>

//Will be mapped 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 "useUnion";
    variant(alt_) "name as ''"
}
```

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

EXAMPLE 3: 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 name="unbounded"/>
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:union>
    </xsd:simpleType>
</xsd:element>

//Will be translated to TTCN-3 as:
type union MaxOccurs {
    XSD.NonNegativeInteger nonNegativeInteger,
    enumerated {unbounded} alt_
}
with {
    variant "name as uncapitalized";
    variant "element";
    variant "useUnion";
    variant(alt_) "name as ''"
}
```

EXAMPLE 4: Mapping member types with enumeration facets applied to different member types:

```
<simpleType name="e22">
    <restriction base="e21unnamed">
        <enumeration value="20"/>
        <enumeration value="50"/>
        <enumeration value="small"/>
    </restriction>
</simpleType>

// will be translated to:
type E21unnamed E22 ({alt_1:=20.0},{alt_1:=50.0},{alt_:="small"})
with {
    variant "name as uncapitalized"
}
```

7.6 ComplexType components

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).

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 (not translated to TTCN-3). 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

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.

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

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

// Will be mapped as:
type record E23
{
    XSD.Integer bar optional,
    XSD.Float foo optional,
    XSD.String base
}
with {
    variant "name as uncapitalized";
    variant(base) "untagged";
    variant(bar, foo) "attribute"
}

// and the template
template E23 t_E23 := {
  bar := 1,
  foo := 2.0,
  base := "something"
}

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

7.6.1.2 Restricting simple content

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: Example for restriction of a base type:

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

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

// and the template
template E24 t_E24 := {
  bar := 1,
  foo := 2.0,
  base := "some"
}

// shall be encoded as:
<?xml version="1.0" encoding="UTF-8"?>
<e23 bar=1 foo=2.0>some</e23>
```

7.6.2 ComplexType containing complex content

In contrast to *simpleContent*, *complexContent* is allowed to have elements. It is possible to extend a base type with 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).

This 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 1: Both the base and the extending types have the compositor *sequence*:

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

<group name="g25seq">
    <sequence>
        <element name="familyStatusElemInGroup" type="string"/>
        <element name="spouseElemInGroup" type="string" minOccurs="0"/>
    </sequence>
</group>

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

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

<!-- Now a type is defined that extends e25seq by adding a new element, group and attributes: -->
<complexType name="e26seq">
    <complexContent>
        <extension base="ns:e25seq">
            <sequence>
                <element name="ageElemExt" type="integer"/>
                <group ref="ns:g25seq"/>
            </sequence>
            <attribute name="unitOfAge" type="string"/>
            <attributeGroup ref="ns:g25attr1"/>
        </extension>
    </complexContent>
</complexType>


// This is translated to the TTCN-3 structure:
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 2: Both the base and the extending types have the compositor *sequence* and multiple occurrences are allowed:

```
<!-- Additional base definition:-->

<complexType name="e25seqRecurrence">
    <sequence minOccurs="0" maxOccurs="unbounded">
        <element name="titleElemBase" type="string"/>
        <element name="forenameElemBase" type="string"/>
        <element name="surnameElemBase" type="string"/>
    </sequence>
    <attribute name="genderAttrBase" type="integer"/>
    <attributeGroup ref="ns:g25attr2"/>
</complexType>

<!-- The extending type definition: -->

<complexType name="e26seqReccurrence">
    <complexContent>
        <extension base="ns:e25seq">
            <sequence minOccurs="0" maxOccurs="unbounded">
                <group ref="ns:g25cho"/>
                <element name="ageElemExt" type="integer"/>
            </sequence>
            <attribute name="unitOfAge" type="string"/>
        </extension>
    </complexContent>
</complexType>

<complexType name="e26seqDoubleRecurrence">
    <complexContent>
        <extension base="ns:e25seqRecurrence">
            <sequence minOccurs="0" maxOccurs="unbounded">
                <group ref="ns:g25cho"/>
                <element name="ageElemExt" type="integer"/>
            </sequence>
            <attribute name="unitOfAge" type="string"/>
        </extension>
    </complexContent>
</complexType>

//The extending types are translated to TTCN-3 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 3: Both the base and the extending types have the compositor *choice*:

```
<complexType name="e25cho">
    <choice>
        <element name="titleElemBase" type="string"/>
        <element name="forenameElemBase" type="string"/>
        <element name="surnameElemBase" type="string"/>
    </choice>
    <attribute name="genderAttrBase" type="string"/>
</complexType>

<!-- and -->
<complexType name="e26cho">
    <complexContent>
        <extension base="ns:e25cho">
            <choice>
                <element name="ageElemExt" type="integer"/>
                <element name="birthdayElemExt" type="date"/>
            </choice>
            <attribute name="unitAttrExt" type="string"/>
        </extension>
    </complexContent>
</complexType>

//Are translated to TTCN-3 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 4: Extension of a *sequence* base type by a *choice* model group:

```
<complexType name="e27cho">
    <complexContent>
        <extension base="ns:e25seq">
            <choice>
                <element name="ageElemExt" type="integer"/>
                <element name="birthdayElemExt" type="date"/>
            </choice>
            <attribute name="unitAttrExt" type="string"/>
        </extension>
    </complexContent>
</complexType>
```

```
// is translated to TTCN-3 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 5: Extending of a base type with *choice* model group by a *sequence* model group:

```
<complexType name="e27seq">
    <complexContent>
        <extension base="ns:e25cho">
            <sequence>
                <element name="ageElemExt" type="integer"/>
            </sequence>
            <attribute name="unitAttrExt" type="string"/>
        </extension>
    </complexContent>
</complexType>

// Is translated to TTCN-3 as:
type record E27seq {
    XSD.String genderAttrBase optional,
    XSD.String unitAttrExt optional,
    union {
        XSD.String ElemBase,
        XSD.String forenameElemBase,
        XSD.String surnameElemBase
    } choice,
    XSD.Integer ageElemExt
}
with {
    variant "name as uncapitalized";
    variant(genderAttrBase, unitAttrExt) "attribute";
    variant(choice) "untagged";
}
```

EXAMPLE 6: Recursive extension of an anonymous inner type is realized using the TTCN-3 dot notation (starts from the name of the outmost type):

```
<complexType name="X">
    <sequence>
        <element name="x" type="string"/>
        <element name="y" minOccurs="0">
            <complexType>
                <complexContent>
                    <extension base="ns:X"/>
                        <sequence>
                            <element name="z" type="string"/>
                        </sequence>
                    </extension>
                </complexContent>
            </complexType>
        </element>
    </sequence>
</complexType>
```

```
// Is translated to the TTCN-3 structure
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 restricts one or more of its components.

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 1: Restricting *anyType*: in the example below *anyType* (any possible type) is used as the base type and it is restricted to only two elements:

```
<complexType name="e28">
    <complexContent>
        <restriction base="anyType">
            <sequence>
                <element name="size" type="nonPositiveInteger"/>
                <element name="unit" type="NMTOKEN"/>
            </sequence>
        </restriction>
    </complexContent>
</complexType>

// Is translated to:
type record E28 {
    XSD.NonPositiveInteger size,
    XSD.NMTOKEN unit
}
with {
    variant "name as uncapitalized"
}
```

EXAMPLE 2: Restricting a user defined complex type (the effect of the *use* attribute is described in clause 7.1.12):

```
<element name="comment" type="string"/>

<!-- The base type is: -->
<complexType name="PurchaseOrderType">
    <sequence>
        <element name="shipTo"     type="string"/>
        <element name="billTo"     type="string"/>
        <element ref="ns:comment" minOccurs="0"/>
        <element name="items"      type="ns:Items"/>
    </sequence>
    <attribute name="shipDate" type="date"/>
    <attribute name="orderDate" type="date"/>
</complexType>

<!-- The restricting type is: -->
<complexType name="RestrictedPurchaseOrderType">
    <complexContent>
        <restriction base="ns:PurchaseOrderType">
            <sequence>
                <element name="shipTo" type="string"/>
                <element name="billTo" type="string"/>
                <element ref="ns:comment" minOccurs="1"/>
                <element name="items"  type="ns:Items"/>
            </sequence>
            <attribute name="shipDate" type="date" use="required" />
            <attribute name="orderDate" type="date" use="prohibited" />
        </restriction>
    </complexContent>
</complexType>
```

```
//is translated to TTCN-3 as:

type XSD.String Comment
with {
    variant "name as uncapitalized";
    variant "element"
}

/* base type */
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"
}

/* restricting type */
type record RestrictedPurchaseOrderType {
    XSD.Date orderDate, //note that this field become mandatory
                                    //note that the field shipDate 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 was 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.3.1.4), 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 <*sequence*>:

```
<!-- Referencing a group with compositor <sequence> (see group declaration in $7.9) -->
<xsd:complexType name="LonelySeqGroup">
    <xsd:group ref="ns:shipAndBill"/>
</xsd:complexType>

//Is translated to TTCN-3 as:
type record LonelySeqGroup {
    XSD.String shipTo,
    XSD.String billTo
}
```

```
<!-- The group reference is optional, compositor <sequence> (see group declaration in $7.9) -->
<xsd:complexType name="LonelySeqGroupOptional">
    <xsd:group ref="ns:shipAndBill" minOccurs="0"/>
</xsd:complexType>

//Is translated to TTCN-3 as:
type record LonelySeqGroupOptional {
    ShipAndBill shipAndBill optional
}

<!-- The group reference is iterative, compositor <sequence> (see group declaration in $7.9) -->
<xsd:complexType name="LonelySeqGroupRecurrence">
    <xsd:group ref="ns:shipAndBill" minOccurs="0" maxOccurs="unbounded"/>
</xsd:complexType>


//Is translated to TTCN-3 as:
type record LonelySeqGroupRecurrence {
    record of ShipAndBill shipAndBill_list
}
with {
    variant (shipAndBill_list) "untagged";
}
```

EXAMPLE 2: Mapping of a group reference, child of *complexType*, compositor <*all*>:

```
<!-- Referencing a group with compositor <all> (see group declaration in $7.9) -->
<xsd:complexType name="LonelyAllGroup">
    <xsd:group ref="ns:shipAndBillAll"/>
</xsd:complexType>

//Is translated to TTCN-3 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 $7.9) -->
<xsd:complexType name="LonelyAllGroupOptional">
    <xsd:group ref="ns:shipAndBillAll" minOccurs="0"/>
</xsd:complexType>

//Is translated to TTCN-3 as:
type record LonelyAllGroupOptional {
    record of enumerated { shipTo, billTo } order,
    XSD.String shipTo optional,
    XSD.String billTo optional
}
with {
    variant "useOrder"
}
```

EXAMPLE 3: Mapping of a group reference, child of *complexType*, compositor <*choice*>:

```
<!-- Referencing a group with compositor <choice> (see group declaration in $7.9) -->
<xsd:complexType name="LonelyChoGroup">
    <xsd:group ref="ns:shipOrBill"/>
</xsd:complexType>

//Is translated to TTCN-3 as:
type record LonelyChoGroup {
    ShipOrBill shipOrBill
}

<!-- The group reference is optional, compositor <choice> (see group declaration in $7.9) -->
<xsd:complexType name="LonelyChoGroupOptional">
    <xsd:group ref="ns:shipOrBill" minOccurs="0"/>
</xsd:complexType>

//Is translated to TTCN-3 as:
type record LonelyChoGroup {
    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>

//Is translated to TTCN-3 as:
type record LonelyChoGroup {
    record of ShipOrBill shipOrBill_list
}
with {
    variant (shipAndBill_list) "untagged";
}
```

EXAMPLE 4: 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>


//Is translated to TTCN-3 as:
type record SeqGroupInSequence {
    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">
        <annotation><documentation xml:lang="EN">sequence group ref.</documentation></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>

//Is translated to TTCN-3 as:
SeqGroupAndElementsAndAttributeInChoice ::= SEQUENCE {
    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 ITU-T Recommendation X.694 [4].

EXAMPLE 1: XSD *all* content model with mandatory elements:

```
<complexType name="e29a">
    <all>
        <element name="foo" type="integer"/>
        <element name="bar" type="float"/>
        <element name="ding" type="string"/>
    </all>
</complexType>

// Is mapped to the following TTCN-3 structure:
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:

```
<complexType name="e29b">
    <all minOccurs="0">
        <element name="foo" type="integer"/>
        <element name="bar" type="float"/>
        <element name="ding" type="string"/>
    </all>
</complexType>

// Is mapped to the following TTCN-3 structure:
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:

```
<complexType name="e29c">
    <all>
        <element name="foo" type="integer"/>
        <element name="bar" type="float" minOccurs="0"/>
        <element name="ding" type="string"/>
    </all>
</complexType>

// Is mapped to the following TTCN-3 structure:
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:

```
<attribute name="attrGlobal" type="token"/>

<attributeGroup name="attrGroup">
    <attribute name="attrInGroup2" type="token"/>
    <attribute name="attrInGroup1" type="token"/>
</attributeGroup>

<complexType name="e29aAndAttributes">
    <all>
        <element name="foo" type="integer"/>
        <element name="bar" type="float"/>
        <element name="ding" type="string"/>
    </all>
    <attribute name="attrLocal" type="integer"/>
    <attribute ref="ns:attrGlobal"/>
    <attributeGroup ref="ns:attrGroup"/>
</complexType>

//Is translated to TTCN-3 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

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).

r) 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 may be changed to "choice_list".

s) 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:

```
<complexType name="e30">
    <choice>
        <element name="foo" type="integer"/>
        <element name="bar" type="float"/>
    </choice>
</complexType>

// Will be translated to:
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*:

```
<group name="e31">
    <sequence>
        <element name="foo" type="string"/>
        <element name="bar" type="string"/>
    </sequence>
</group>

<complexType name="e32">
    <choice>
        <group ref="ns:e31"/>
        <element name="ding" type="string"/>
    </choice>
</complexType>

//Is translated to TTCN-3 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:

```
<complexType name="e33">
    <choice>
        <choice>
            <element name="foo" type="string"/>
            <element name="bar" type="string"/>
        </choice>
        <element name="ding" type="string"/>
    </choice>
</complexType>

// Is mapped to TTCN-3 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.

EXAMPLE 1: Single *sequence* nested to *choice*:

```
<complexType name="e34a">
    <choice>
        <sequence>
            <element name="foo" type="string"/>
            <element name="bar" type="string"/>
        </sequence>
        <element name="ding" type="string"/>
    </choice>
</complexType>

// Is translated to:
```

```
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*:

```
<complexType name="e34b">
    <choice>
        <sequence>
            <sequence>
                <element name="foo" type="string"/>
                <element name="bar" type="string"/>
            </sequence>
            <element name="ding" type="string"/>
            <element name="foo" type="string"/>
            <element name="bar" type="string"/>
        </sequence>
        <element name="ding" type="string"/>
    </choice>
</complexType>

// Is translated to:
type record E34b {
    union {
        record {
            record {
                XSD.String foo,
                XSD.String bar
            } sequence,
            XSD.String ding,
            XSD.String foo,
            XSD.String bar
        } sequence,
        XSD.String ding
    } choice
}
with {
    variant "name as uncapitalized ";
    variant(choice, choice.sequence, choice.sequence.sequence) "untagged"
}
```

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:

```
<complexType name="e35">
    <choice>
        <element name="foo" type="string"/>
        <any namespace="other"/>
    </choice>
</complexType>

// Is translated to:
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

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:

```
<complexType name="e36a">
    <sequence>
        <element name="foo" type="integer"/>
        <element name="bar" type="float"/>
    </sequence>
</complexType>

// Is mapped to
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*:

```
<group name="e37">
    <choice>
        <element name="foo" type="string"/>
        <element name="bar" type="string"/>
    </choice>
</group>

<complexType name="e38">
    <sequence>
        <group ref="ns:e37"/>
        <element name="ding" type="string"/>
    </sequence>
</complexType>

// Is translated to:
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:

```
<complexType name="e39">
    <sequence>
        <choice>
            <element name="foo" type="string"/>
            <element name="bar" type="string"/>
        </choice>
        <element name="ding" type="string"/>
    </sequence>
</complexType>

// Is translated to:
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, a *sequence* nested in a *sequence* shall be translated to TTCN-3 according to clause 7.6.6 and the resulted constructs shall be added to the enframing **record** type or field (see also clauses 7.6 and 7.6.5.4).

EXAMPLE 1: Sequence nesting a mandatory *sequence*:

```
<complexType name="e40a">
    <sequence>
        <sequence>
            <element name="foo" type="string"/>
            <element name="bar" type="string"/>
        </sequence>
        <element name="ding" type="string"/>
    </sequence>
</complexType>

// Is mapped 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:

```
<complexType name="e40b">
    <sequence>
        <sequence>
            <element name="foo" type="string"/>
            <element name="bar" type="string"/>
        </sequence>
        <choice>
            <element name="foo" type="string"/>
            <element name="bar" type="string"/>
        </choice>
        <element name="ding" type="string"/>
```

```
    </sequence>
</complexType>

// Is mapped 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:

```
<complexType name="e41">
    <sequence>
        <element name="foo" type="string"/>
        <any/>
    </sequence>
</complexType>

// Is translated to:
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".

t) The encoding instruction "untagged" shall be attached to the field corresponding to *sequence*.

u) 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 may be changed to "sequence_list".

v) 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*:

```
<complexType name="e36b">
    <sequence minOccurs="0">
        <element name="foo" type="integer"/>
        <element name="bar" type="float"/>
    </sequence>
</complexType>

// Is mapped to
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:

```
<complexType name="e40c">
    <sequence>
        <sequence minOccurs="0">
            <element name="foo" type="string"/>
            <element name="bar" type="string"/>
        </sequence>
        <choice>
            <element name="foo" type="string"/>
            <element name="bar" type="string"/>
        </choice>
        <element name="ding" type="string"/>
    </sequence>
</complexType>

// Is mapped to
type record E40c {
    record {
        XSD.String foo,
        XSD.String bar
    } sequence optional,
    union {
        XSD.String foo,
        XSD.String bar
    } choice,
    XSD.String ding
}
with {
    variant "name as uncapitalized";
    variant(sequence, choice) "untagged"
}
```

EXAMPLE 3: Sequence nesting a sequence of multiple recurrence:

```
<complexType name="e40d">
    <sequence>
        <sequence minOccurs="0" maxOccurs="unbounded">
            <element name="foo" type="string"/>
            <element name="bar" type="string"/>
        </sequence>
        <element name="ding" type="string"/>
    </sequence>
</complexType>

// Is mapped to
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"
}
```

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, 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.3.1.4), 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*:

```
<attributeGroup name="e42">
    <attribute name="foo" type="float"/>
    <attribute name="bar" type="float"/>
</attributeGroup>

<complexType name="e44">
    <sequence>
        <element name="ding" type="string"/>
    </sequence>
    <attributeGroup ref="ns:e42"/>
</complexType>

// Is translated to TTCN-3 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>
```

```
//is translated to TTCN-3 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"
  //Please note, the order of the field names in the attribute qualifier may be arbitrary
}
```

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"
  //Please note, the order of the field names in the attribute qualifier may be arbitrary
}
```

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:

```
<element name="MySeqMixed">
    <xsd:complexType name="MyComplexType-12" mixed="true">
        <xsd:sequence>
            <xsd:element name="a" type="xsd:string"/>
            <xsd:element name="b" type="xsd:boolean"/>
        </xsd:sequence>
        <attribute name="attrib" type="integer"/>
    </xsd:complexType>
</element>

// Is translated to the TTCN-3 type definition
type record MySeqMixedMyComplexType_12 {
    record of XSD.String    embed_values,
    // in TTCN-3 values the embed_values field may have max. 3 record of components
    XSD.Integer attrib optional,
    XSD.String a,
    XSD.Boolean b
}
with {
    variant "element";
    variant "embedValues";
    variant(attrib) "attribute"
}

//And the template
template MySeqMixedMyComplexType_12 t_MySeqMixedMyComplexType_12 := {
    embed_values:= {"The ordered", "has arrived", "Wait for further information."},
    a:= "car",
    b:= true
}

//will be encoded as
<MySeqMixedMyComplexType-12>
    The ordered
    <a>car</a>
    has arrived
    <b>true</b>
    Wait for further information.
</MySeqMixedMyComplexType-12>
```

EXAMPLE 2: Complex type definition with *sequence* constructor of multiple occurrences and *mixed* content type:

```
<element name="MyComplexElem-16">
    <xsd:complexType name="MyComplexType-16" 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>
</element>
```

```
// Is translated to the TTCN-3 type definition
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 as
<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.
</MyComplexTypeElem-16>
```

EXAMPLE 3: Complex type definition with *all* constructor and *mixed* content type:

```
<element name="MyComplexElem-13">
    <xsd:complexType name="MyComplexType-13" mixed="true">
        <xsd:all>
            <xsd:element name="a" type="xsd:string"/>
            <xsd:element name="b" type="xsd:boolean"/>
        </xsd:all>
    </xsd:complexType>
</element>

// Is translated to the TTCN-3 type definition
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 as
<MyComplexTypeElem-13>
    Arrival status
    <b>false</b>
    product name
    <a>car</a>
    Wait for further information.
</MyComplexTypeElem-13>
```

EXAMPLE 4: Complex type definition with *all* constructor, optional elements and *mixed* content type:

```
<xsd:complexType name="MyComplexType-15" mixed="true">
    <xsd:all minOccurs="0">
        <xsd:element name="a" type="xsd:string"/>
        <xsd:element name="b" type="xsd:boolean"/>
    </xsd:all>
</xsd:complexType>

// Is translated to the TTCN-3 type definition
type record MyComplexType_15 {
    record of XSD.String embed_values,
    record of enumerated {a,b} order,
    XSD.String a optional,
    XSD.Boolean b optional
}
with {
    variant "embedValues";
    variant "useOrder"
}

//And the template
template MyComplexType_15 t_MyComplexType_15 := {
    embed_values:= {"Arrival status", "Wait for further information."},
    order := {b},
    a:= omit,
    b:= false
}
//will be encoded as
<MyComplexType-15>
    Arrival status
    <b>false</b>
    Wait for further information.
</MyComplexType-15>
```

EXAMPLE 5: Complex type definition with *choice* constructor and *mixed* content type:

```
<element name="MyComplexElem-14">
    <xsd:complexType name="MyComplexType-14" mixed="true">
        <xsd:choice>
            <xsd:element name="a" type="xsd:string"/>
            <xsd:element name="b" type="xsd:boolean"/>
        </xsd:choice>
    </xsd:complexType>
</element>

// Is translated to the TTCN-3 type definition
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 as
<MyComplexTypeElem-14>
    Arrival status
    <b>false</b>
    Wait for further information.
</MyComplexTypeElem-14>
```

7.7 Any and anyAttribute

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. The content of this XML shall not be parsed and interpreted by the encoder and decoder. 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.

The *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` and the name of the field shall be the result of applying clause 5.2.2 to "elem".

NOTE: The mapping may be influenced by the attributes applied to the component, if any. See more details in clause 7.4, especially clause 7.1.4.

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 of`** `XSD.String` 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* components (e.g. by a complex type extending an another complex type containing already *anyAttribute*), only one **`record of`** `XSD.String` field shall be generated (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 codec shall be controlled by an "anyElement " or "anyAttributes"" encoding instructions correspondingly, complemented with an optional "from" or "except" clause specifying the list of namespaces which are allowed or restricted to qualify the given attribute or element. Details on constructing and use of the "anyAttributes" and "anyElement" encoding instructions are given in clauses B.3.3 and B.3.2, correspondingly.

EXAMPLE 1: Translating *any:*

```
<!--
    Please note, the target namespace of the complexType definitions below is
    "http://www.example.org/ttcn/wildcards"
-->

<xs:complexType name="e46">
    <xs:sequence>
        <xs:any namespace="##any"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="e46a">
    <xs:sequence>
        <xs:any minOccurs="0" namespace="##other"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="e46b">
    <xs:sequence>
        <xs:any minOccurs="0" maxOccurs="unbounded" namespace="##local"/>
    </xs:sequence>
</xs:complexType>

//Are mapped to the following TTCN-3 code and encoding extensions:
type record E46 {
    XSD.String  elem
}
with {
    variant "name as uncapitalized";
    variant(elem) "anyElement"
}

type record E46a {
```

```
    XSD.String  elem optional
}
with {
    variant "name as uncapitalized";
    variant(elem) "anyElement except unqualified,'http://www.organization.org/ttcn/wildcard'"
}

type record E46b {
    record of XSD.String elem_list
}
with {
    variant "name as uncapitalized";
    variant(elem_list) "anyElement except unqualified"
}
```

EXAMPLE 2: Translating *anyAttribute*:

```
<!-- Please note, the target namespace of the complexType definitions below is
"http://www.example.org/ttcn/wildcards" -->
<xs:complexType name="e45">
    <xs:anyAttribute namespace="##any"/>
</xs:complexType>

<xs:complexType name="e45a">
    <xs:anyAttribute namespace="##other"/>
</xs:complexType>


<xs:complexType name="e45b">
    <xs:anyAttribute namespace="##targetNamespace"/>
</xs:complexType>

<xs:complexType name="e45c">
    <xs:anyAttribute namespace="##local http://www.example.org/ttcn/attribute"/>
</xs:complexType>

<xs:complexType name="e45d">
    <xs:complexContent>
        <xs:extension base="e45c">
            <xs:anyAttribute namespace="##targetNamespace"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>


//Are mapped as follows:
type record E45 {
    record of XSD.String attr
}
with {
    variant "name as uncapitalized";
    variant(attr) "anyAttributes"
}

type record E45a {
    record of XSD.String attr
}
with {
    variant "name as uncapitalized";
    variant(attr) "anyAttributes except 'http://www.example.org/ttcn/wildcards'"
}

type record E45b {
    record of XSD.String attr
}
with {
    variant "name as uncapitalized";
    variant(attr) "anyAttributes from 'http://www.example.org/ttcn/wildcards'"
}

type record E45c {

with {
    variant "name as uncapitalized";
    variant(attr) "anyAttributes from unqualified,'http://www.example.org/ttcn/attribute'"
}
```

```
type record E45d {
    record of XSD.String attr
}
with {
    variant "name as uncapitalized";
    variant(attr) "anyAttributes from unqualified, 'http://www.example.org/ttcn/attribute',
'http://www.example.org/ttcn/wildcards'"
}
```

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:

```
<annotation>
    <appinfo>Note</appinfo>
    <documentation xml:lang="en">This is a helping note!</documentation>
</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 the same way as *complexTypes* with one difference: the "untagged" encoding instruction shall be attached to the generated TTCN-3 component, corresponding to the *group* element (but, of course, will always represent a subset of complex types, as only the above compositors are allowed as child and can never have attributes).

EXAMPLE: Mapping of groups:

```
<xs:group name="shipAndBill">
  <xs:sequence>
    <xs:element name="shipTo" type="xs:string"/>
    <xs:element name="billTo" type="xs:string"/>
  </xs:sequence>
</xs:group>

<xs:group name="shipOrBill">
  <xs:choice>
    <xs:element name="shipTo" type="xs:string"/>
    <xs:element name="billTo" type="xs:string"/>
  </xs:choice>
</xs:group>

<xs:group name="shipAndBillAll">
  <xs:all>
    <xs:element name="shipTo" type="xs:string"/>
    <xs:element name="billTo" type="xs:string"/>
  </xs:all>
</xs:group>

//Is translated to TTCN-3 as:
type record ShipAndBill {
    XSD.String shipTo,
    XSD.String billTo
}
with {
    variant "untagged"
}
```

```
type record 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 are 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.

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 ITU-T Recommendation 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 ITU-T Recommendation X.694 [4], then transformations described in ES 201 873-7 [2]. 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(0(0[1-9]|[1-9][0-9])|[1-9][0-9][0-9])|[1-9][0-9][0-9][0-9])",
  yearExpansion := "(-([1-9][0-9]#(0,))#(,1))#(,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 String attr,
record of String elem_list
} with {

  variant "XSD:anyType";
  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 with {
         variant "XSD:decimal"
    };

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

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

    // Time types

type charstring  Duration (pattern ") with {
         variant "XSD:duration"
    };
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(0,0,0,12)..char(0,0,0,12),
  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

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.

NOTE: 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.

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" paragraph 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 The XML encode attribute

The encode attribute "XML" shall be used to identify:

```
Syntactical structure
    encode """ (XML | XML1.0 | XML1.1 ) """
Applicable to (TTCN-3)
Module, group, definition.
```

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).

NOTE: 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 shall be applied to the XSD.String type as well, like in all other cases. See for example the conversion of XSD complex type e46b in clause 7.7.

Description

One TTCN-3 attribute 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 wildcard. 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 "*<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 >*"	

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).

Description

Except its applicability it shall be constructed and used the same way as the anyElement encoding instruction (see clause B.3.2).

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

The control namespace is the namespace to be used for the type identification attributes and schema instances (e.g. in the special XML attribute value "xsi:nil", see mapping of the *nillable* XSD attribute in clause 7.1.11). It shall be specified globally, with an encoding instruction attached to the TTCN-3 module.
The first *freetext* component identifies the namespace (normally "http://www.w3.org/2001/XMLSchema-instance" is used), the second *freetext* component identifies the namespace prefix (normally "xsi" is used).

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 or element is omitted in the received XML document.

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

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 field of the record (corresponding to the nillable element) 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 optional field of the record in the corresponding TTCN-3 value shall be omitted. 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 initiate a test case error.

B.3.16 Use Union

Syntactical structure(s)

variant """ useUnion """

Applicable to (TTCN-3)

Types and field of structured types generated for XSD *simpleTypes* derived by *union* (see clause 7.5.3).

Description

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, identifying the XSD base datatype of the selected alternative, except when encoding attributes or the encoded component has a "list" encoding instruction attached. 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.

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`** encoding instruction) 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`** encoding instruction) 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.

Shall not be applied to TTCN-3 components generated for XSD attribute elements (neither global nor local).

For typical use in case of extending or restricting simple content see clauses 7.6.1.1 and 7.6.1.2 and for typical use in case of model groups see clause 7.9.

NOTE: Please note, that using the **`untagged`** encoding instruction in other cases than specified in the present document, may result in an undecodable XML document.

Annex C (informative): Examples

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

XML Schema:

```
<xs:schema xmlns:xs="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.-->

    <xs:element name="shiporder">
        <xs:complexType>
            <xs:sequence>

                <xs:element name="orderperson" type="xs:string"/>

                <xs:element name="shipto">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="name" type="xs:string"/>
                            <xs:element name="address" type="xs:string"/>
                            <xs:element name="city" type="xs:string"/>
                            <xs:element name="country" type="xs:string"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>

                <xs:element name="item" >
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="title" type="xs:string"/>
                            <xs:element name="note" type="xs:string" minOccurs="0"/>
                            <xs:element name="quantity" type="xs:positiveInteger"/>
                            <xs:element name="price" type="xs:decimal"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>

            </xs:sequence>
            <xs:attribute name="orderid" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

TTCN-3 Module:

```
module Example1 {

import from XSD language "XML" all;


type record Shiporder {
  XSD.String orderid,
  XSD.String orderperson,
  record
  {
   XSD.String name,
   XSD.String address_1,
   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(shipto.address_1)"name as 'address'";
 variant(orderid) "attribute";
}


} with {
encode "XML";
}

   module Example1Template {

import from XSD language "XML" all;
import from Example1 all;



template Shiporder t_Shiporder:={
  orderid:="18920320_17",
  orderperson:="Dr.Watson",
  shipto:=
  {
   name:="Sherlock Holmes",
   addressField:="Baker Street 221B",
   city:="London",
   country:="England"
  },
  item:=
  {
   title:="Memoirs",
   note:= omit,
   quantity:=2,
   price:=3.5
  }
}


}//end module

<?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>3.5</price>
</item>
</shiporder>
```

C.2 Example 2

XML Schema:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:simpleType name="S1">
        <xs:restriction base="xs:integer">
            <xs:maxInclusive value="2"/>
        </xs:restriction>
    </xs:simpleType>
```

```
    <xs:simpleType name="S2">
        <xs:restriction base="S1">
            <xs:minInclusive value="-23"/>
            <xs:maxInclusive value="1"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="S3">
        <xs:restriction base="S2">
            <xs:minInclusive value="-3"/>
            <xs:maxExclusive value="1"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="C1">
        <xs:simpleContent>
            <xs:extension base="S3">
                <xs:attribute name="A1" type="xs:integer"/>
                <xs:attribute name="A2" type="xs:float"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>

</xs:schema>
```

TTCN-3 Module:

```
module Example2 {

    import from XSD language "XML" all;

    type XSD.Integer S1 (-infinity .. 2);

    type S1 S2 (-23 .. 1);

    type S2 S3 (-3 .. 0);

    type record C1 {
        S3              base,
        XSD.Integer     a1 optional,
        XSD.Float       a2 optional
    } with {
     variant(a1,a2) "name as capitalized ";
     variant(a1,a2) "attribute";
   variant(base) "untagged"
    }
} with {
encode "XML";
}


module Example2Templates {

    import from XSD language "XML" all;
    import from Example2 all;

    template C1 t_C1:= {
        base    :=-1,
        a1  :=1,
        a2  :=2.0
    }
}


<?xml version="1.0" encoding="UTF-8"?>
<C1 A1="1" A2="2.0">-1</C1>
```

C.3 Example 3

XML Schema:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="nsA" targetNamespace="nsA">

    <xs:complexType name="C1">
        <xs:simpleContent>
            <xs:extension base="xs:integer">
                <xs:attribute name="A1" type="xs:integer"/>
                <xs:attribute name="A2" type="xs:integer"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>

    <xs:complexType name="C2">
        <xs:simpleContent>
            <xs:restriction base="C1">
                <xs:minInclusive value="23"/>
                <xs:maxInclusive value="26"/>
                <xs:attribute name="A1" type="xs:byte" use="required"/>
                <xs:attribute name="A2" type="xs:negativeInteger"/>
            </xs:restriction>
        </xs:simpleContent>
    </xs:complexType>

    <xs:complexType name="C3">
        <xs:simpleContent>
            <xs:restriction base="C2">
                <xs:minInclusive value="25"/>
                <xs:maxInclusive value="26"/>
            </xs:restriction>
        </xs:simpleContent>
    </xs:complexType>

</xs:schema>
```

TTCN-3 Module:

```
module Example3 {

    import from XSD language “XML” all;

    type record C1 {
        XSD.Integer base,
        XSD.Integer a1 optional,
        XSD.Integer a2 optional
    } with {
      variant(a1,a2)"name as capitalized";
      variant(a1,a2) "attribute";
   variant(base) "untagged"
   }



    type record C2 {
        XSD.Integer (23 .. 26) base,
        XSD.Byte        a1,
        XSD.NegativeInteger  a2 optional
    } with {
      variant(a1,a2)"name as capitalized";
      variant(a1,a2) "attribute";
   variant(base) "untagged" ;
  }


    type record C3 {
        XSD.Integer (25 .. 26) base,
        XSD.Byte        a1,
        XSD.NegativeInteger  a2 optional
    } with {
      variant(a1,a2)"name as capitalized";
      variant(a1,a2) "attribute";
   variant(base) "untagged"
    }
```

```
} with {
encode "XML";

variant "namespace as 'nsA'"
}


module Example3Templates {

import from XSD language "XML" all;
import from Example3 all;

    template C1 t_C1:= {
        base    :=-1000,
        a1  :=1,
        a2  :=2
    }


    template C2 t_C2:= {
        base    :=24,
        a1  :=1,
        a2  :=-2
    }


    template C3 t_C3:= {
        base    :=25,
        a1  :=1,
        a2  :=-1000
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<C1 xmlns="nsA" A1=1 A2=2>-1000</C1>


<?xml version="1.0" encoding="UTF-8"?>
<C2 xmlns="nsA" A1=1 A2=-2>24</C2>

<?xml version="1.0" encoding="UTF-8"?>
<C3 xmlns="nsA" A1="1" A2="-1000">25</C3>
```

C.4 Example 4

XML Schema:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:NA="nsA" targetNamespace="nsA">

    <xs:include schemaLocation="Example3.xsd"/>
    <xs:import schemaLocation="Example2.xsd"/>

    <xs:complexType name="newC1">
        <xs:complexContent>
            <xs:extension base="NA:C1"/>
        </xs:complexContent>
    </xs:complexType>

    <xs:simpleType name="newS1">
        <xs:restriction base="S1"/>
    </xs:simpleType>

</xs:schema>
```

TTCN-3 Module:

```
module Example4 {

    import from XSD language "XML" all;
    import from Example2 language "XML" all;
    import from Example3 language "XML" all;
```

```
    type Example3.C1 NewC1
    with {variant "name as uncapitalized"}

    type Example2.S1 NewS1
    with {variant "name as uncapitalized"}

} with {
encode "XML";
variant "namespace as 'nsA' prefix 'NA'"
}


    module Example4Templates {


    import from XSD language "XML" all;
    import from Example2 language "XML" all;
    import from Example3 language "XML" all;
    import from Example4 all;

    template NewC1 t_NewC1:= {
        base    :=-1000,
        a1  :=1,
        a2  :=2
    }

    template NewS1 NewS1:=1
}

<?xml version="1.0" encoding="UTF-8"?>
<NA:newC1 xmlns:NA="nsA" A1="1" A2="2">-1000</NA:newC1>

<?xml version="1.0" encoding="UTF-8"?>
<NA:newS1 xmlns:NA="nsA">1</NA:newS1>
```

Annex D (informative): Deprecated features

D.1 Using the anyElement encoding instruction to record of fields

The TTCN-3 core language, ES 201 873-1 [1], up to and including version 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 this document.

EXAMPLE 1: The outdated syntax:

```
<xs:complexType name="e46b">
    <xs:sequence>
        <xs:any minOccurs="0" maxOccurs="unbounded" namespace="##local"/>
    </xs:sequence>
</xs:complexType>

//Was mapped to the following TTCN-3 code and encoding extensions according to
//elder versions of this 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:

```
<xs:complexType name="e46b">
    <xs:sequence>
        <xs:any minOccurs="0" maxOccurs="unbounded" namespace="##local"/>
    </xs:sequence>
</xs: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
}
```

History

Document history		
V3.3.1	July 2008	Publication
V4.1.1	June 2009	Publication
V4.2.1	May 2010	Membership Approval Procedure MV 20100723: 2010-05-24 to 2010-07-23
V4.2.1	July 2010	Publication