ETSI ES 201 873-9 V3.3.1 (2008-07)

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

DES/MTS-00090-9 ttcn3 & XML

Keywords

MTS, testing, TTCN, XML

Contents

Intellectual Property Rights

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 [8] and [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 a specific reference, subsequent revisions do not apply.
- Non-specific reference may be made only to a complete document or a part thereof and only in the following cases:
  - if it is accepted that it will be possible to use all future changes of the referenced document for the purposes of the referring document;
  - for informative references.

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

For online referenced documents, information sufficient to identify and locate the source shall be provided. Preferably, the primary source of the referenced document should be cited, in order to ensure traceability. Furthermore, the reference should, as far as possible, remain valid for the expected life of the document. The reference shall include the method of access to the referenced document and the full network address, with the same punctuation and use of upper case and lower case letters.

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 indispensable for the application of the present document. For dated references, only the edition cited applies. For non-specific references, the latest edition of the referenced document (including any amendments) applies.

[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 (07/2002): "Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation".

[4] ITU-T Recommendation X.694 (01/2004): "Information technology - ASN.1 encoding rules: Mapping W3C XML schema definitions into ASN.1".

[5] W3C Recommendation (2004): "Extensible Markup Language (XML) 1.1", World Wide Web Consortium.

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

[6] W3C Recommendation (1999): "Namespaces: Namespaces in XML", World Wide Web Consortium.

NOTE: Available at http://www.w3.org/TR/1999/REC-xml-names-19990114, http://www.w3.org/XML/Schema

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

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

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

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

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

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

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

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

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

[12] W3C Information technology: "ASN.1 encoding rules - XML encoding rules ITU-T Recommendation X.693".

[13] W3C Information technology: "ASN.1 encoding rules - XML encoding rules Amendment 1 to ITU-T Recommendation X.693".

2.2 Informative references

The following referenced documents are not essential to the use of the present document but they assist the user with regard to a particular subject area**.** For non-specific references, the latest version of the referenced document (including any amendments) applies.

Not applicable.

3 Symbols and abbreviations

3.1 Symbols

For the purposes of the present document, the following symbols aply:

␣	The whitespace character

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	IETF Uniform Resource Identifier
W3C	World Wide Web Consortium
XML	W3C Extensible Markup Language
XSD	W3C 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 [10] 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.

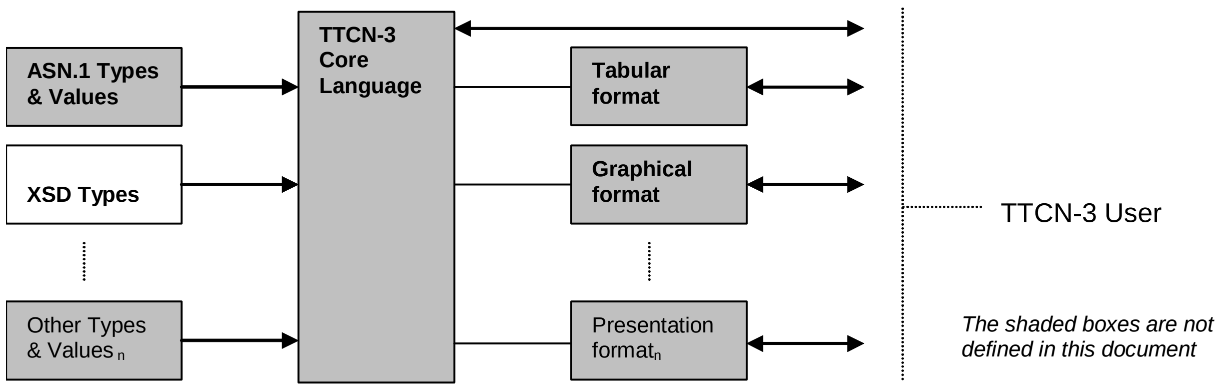

Figure 1: User's view of the core language and the various presentation formats

For compatibility reasons, the TTCN-3 code obtained from the XML Schema using the present document for an explicit mapping should 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 should 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.

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 is produced from the XML Schema, which representation should 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 is 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 will have to import the XSD.ttcn module (see annex A) .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.

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 will be translated to one ore more TTCN-3 modules, corresponding to schema components that have the same target namespace. Any XSD *include* / *import* statements are mapped to equivalent TTCN-3 *import* statements. An XSD Schema including another XSD Schema will be translated to two TTCN-3 modules, one importing from the other, both having the same target namespace. The module names generated by the translation are not standardized.

As TTCN-3 does not offer a namespace concept, information about namespaces and prefixes (from targetNamespace, elementFormDefault, attributeFormDefault, etc.) will be preserved not in the TTCN-3 code but in the encoding extensions (internal or external). To allow this, an extension permitting explicit specification of the namespaces and prefixes is introduced:

```
module MyModule
{
:
} with {
encode "XML";
variant "namespace all, all in all as 'http://www.example.org/' prefix 'ns0'"
}
```

All types declared in the module will inherit the namespace declaration of the module. This can be overridden by namespace declarations qualifying fields of declared structures:

```
Example:

module MyModule
{
:

type record MyRecordType
{
integer        field1,
charstring     field2
}
with {
variant (field1) "namespace as 'http://www.example.org/example1' prefix 'ns1'";
}
:

} with {
encode "XML";
variant "namespace all, all in all as 'http://www.example.org/' prefix 'ns0'"
}



template MyRecordType MyTemplate:=
{
field1:= 3,
field2:= "four"
}
```

will be encoded as:

```
<?xml version="1.0" encoding="UTF-8"?>
<ns0:MyRecordType xmlns:ns0=http://www.example.org/>
< ns1:field1 xmlns:ns1=http://www.example.org/example1>3</ns1:field1>
< ns0:field2>four</ ns0:field2>
</ns0:MyRecordType>
```

If a module has no namespace declaration, all types and fields of types declared within the module are assumed to have no namespace, except when a qualifying statement refers a declared type or field directly.
The importation structure of the TTCN-3 modules will retain the importation structure of the imported XML schemas. TTCN-3 'import' statements will import types declared in other modules, and referenced in the current module.
The control namespace (the namespace of the type identification attributes and of the nil identification attribute) will be specified globally, with an encoding extension attached to the TTCN-3 module:

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

Qualifying declarations of the namespace prefixes can be used against templates, allowing re-declaration of namespace prefixes inherited from the referred types:

```
template MyRecordType  MyTemplate2:=
{
field1:= 3,
field2:= "four"
} with {
encode "XML";
variant (field1) "namespace prefix 'newns1'";
variant (field2) "namespace prefix 'newns0'"

}
```

will be encoded as:

```
<?xml version="1.0" encoding="UTF-8"?>
<newns0:MyRecordType xmlns:newns0=http://www.example.org/>
<newns1:field1 xmlns:newns1=http://www.example.org/example1>3</newns1:field1>
<newns0:field2>four</newns0:field2>
</newns0:MyRecordType>
```

For reception templates, wildcards (*,?) are allowed to be used in the prefix declaration.

For sake of clarity, wherever irrelevant, namespaces have been omitted from most of the examples in the present document.

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 valid in TTCN-3. For example it would be fine 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 ("base" or "content").

The present document specifies the generation of:

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

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

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

d) TTCN-3 type reference names of special type assignments (Generating special ASN.1 type assignments for element declarations, Generating special ASN.1 type assignments for type definitions and Generating special ASN.1 type assignments for element substitution groups); and

e) TTCN-3 identifiers of certain sequence components introduced by the mapping (see clause 20).

All of these TTCN-3 names are 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 Identifier 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 or can contain characters not allowed in TTCN-3 identifiers or in TTCN-3 type reference names. 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.

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

- the characters " " (SPACE) and "`.`" (FULL STOP) shall all be replaced by a "_" (LOW LINE); and
- 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; and
- a sequence of two or more "_" (LOW LINE) characters shall be replaced with a single "_" (LOW LINE); and
- "_" (LOW LINE) characters occurring at the beginning or at the end of the name shall be removed; and
- if a character string that is to be used as a type reference name 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; and
- if a character string that is to be used as an identifier 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; and
- if a character string that is to be used as a type reference name is empty, it shall be replaced by "**X**" (LATIN CAPITAL LETTER X); and
- if a character string that is to be used as an identifier is empty, it shall be replaced by "**x**" (LATIN SMALL LETTER X).

Depending on the kind of name being generated, one of the three following items shall apply.

a) If the name being generated is the type reference name of an TTCN-3 type assignment and the character string generated by 10.3.3 is identical to the type reference name of another TTCN-3 type assignment previously generated in the same TTCN-3 module or in another TTCN-3 module with the same namespace (including absence of a namespace), or is one of the reserved words specified in annex A of ES 201 873-1 [1], in clause 11.27 of ITU-T Recommendation X.680 [3], then a suffix shall be appended to the character string generated according to the above rules. The suffix shall consist of a "_" (LOW LINE) followed by the canonical lexical representation (see W3C XML Schema Part 2 [9], clause 2.3.1) of an integer. This integer shall be the least positive integer such that the new name is different from the type reference name of any other TTCN-3 type assignment previously generated in any of those TTCN-3 modules.

b) If the name being generated is the identifier of a field of a record, a set or a union type, and the character string generated by the above rules is identical to the identifier of a previously generated field identifier of the same type, then a suffix shall be appended to the character string generated by the above rules. The suffix shall consist of a "_" (LOW LINE) followed by the canonical lexical representation (see W3C XML Schema Part 2 [9], clause 2.3.1) of an integer. This integer shall be the least positive integer such that the new identifier is different from the identifier of any previously generated component of that sequence, set, or choice type.

c) 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 above rules is identical to the identifier of another enumeration item previously generated in the same enumerated type, then a suffix shall be appended to the character string generated by the above rules. The suffix shall consist of a "_" (LOW LINE) followed by the canonical lexical representation (see W3C XML Schema Part 2 [9], clause 2.3.1) of an integer. This integer shall be the least positive integer such that the new identifier is different from the identifier in any other enumeration item already present in that TTCN-3 enumerated type.

For an TTCN-3 type reference name (or 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 reference name (or identifier) generated is different from the **name**, a final **NAME** variant attribute shall be assigned to the TTCN-3 type definition with that type reference 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 reference 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 tye!).

c) Otherwise, the "**name as '*name*'**" variant attribute shall be used.

EXAMPLE 1:

```
//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 this clause for the mapping of a simple type definition with an enumeration facet where the identifier generated is different from the corresponding member of the **value** of the **enumeration** facet, a 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) Otherwise, the "text 'TTCN-3 enumeration identifier' as 'member of the ***value*** of the ***enumeration*** facet'" variant attribute shall be used.

EXAMPLE 2:

```
//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 is 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 lexicographical 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 lexicographical order.

The mapping generates some TTCN-3 type definitions that do not correspond directly to any XSD schema component. These are:

a) union types (with a final **`use_type`** variant attribute) corresponding to a type derivation hierarchy; the type reference names of these types have a "_derivations" suffix;

b) union types (with a final **`use_type`** variant attribute on the type and a final **`use_nil`** variant attribute on each alternative) corresponding to a type derivation hierarchy where the user-defined top-level simple type definition or complex type definition that is the root of the derivation hierarchy is used as the type definition of one or more element declarations that are nillable; the type reference names of these types have a "_deriv_nillable" suffix;

c) union types (with a final **`use_type`** variant attribute on the type and a final **`default_for_empty`** variant attribute on each alternative) corresponding to a type derivation hierarchy where the user-defined top-level simple type definition or complex type definition that is the root of the derivation hierarchy is used as the type definition of one or more element declarations that are not nillable and have a value constraint that is a default value; the type reference names of these types have a "_deriv_default_" suffix;

d) union types (with a final **`use_type`** variant attribute on the type and a final **`default_for_empty`** variant attribute on each alternative) corresponding to a type derivation hierarchy where the user-defined top-level simple type definition or complex type definition that is the root of the derivation hierarchy is used as the type definition of one or more element declarations that are not nillable and have a value constraint that is a fixed value; the type reference names of these types have a "_deriv_fixed_" suffix;

e) union hoice types (with a final **`use_type`** variant attribute on the type and final **`use_nil`** and **`default_for_empty`** variant attribute on each alternative) corresponding to a type derivation hierarchy where the user-defined top-level simple type definition or complex type definition that is the root of the derivation hierarchy is used as the type definition of one or more element declarations that are nillable and have a value constraint that is a default value; the type reference names of these types have a "_deriv_nillable_default_" suffix;

f) union types (with a final **`use_type`** variant attribute on the type and final **`use_nil`** and **`default_for_empty`** encoding instructions on each alternative) corresponding to a type derivation hierarchy where the user-defined top-level simple type definition or complex type definition that is the root of the derivation hierarchy is used as the type definition of one or more element declarations that are nillable and have a value constraint that is a fixed value; the type reference names of these types have a "_deriv_nillable_fixed_" suffix;

g) union types (with a final **`untagged`** variant attribute) corresponding to an element substitution group; the type reference names of these types have a "_group" suffix;

h) record types (with a final `use_nil` variant attribute) corresponding to the use of a user-defined top-level simple type definition or complex type definition as the type definition of one or more element declarations that are nillable; the type reference names of these types have a "_nillable" suffix.

All TTCN-3 type assignments that correspond directly to the XSD schema components in the source XSD Schema shall be generated before all TTCN-3 type assignments listed in clause 10.4.3 (if any).

TTCN-3 type assignments that correspond directly to the XSD schema components shall be generated in the order of the corresponding XSD schema components. TTCN-3 type definitions listed in items a) to h) above (if any) shall be generated in the order of the XSD schema components corresponding to the "associated type definition".

For items c) to f) above, if the simple type definition or complex type definition that is the root of the derivation hierarchy is used as the type definition of multiple element declarations that have different values in the value constraint, the TTCN-3 type definitions shall be generated in ascending lexicographical order of the canonical lexical representation (see W3C XML Schema Part 2 [9], clause 2.3.1) of the value in the value constraint.

NOTE 2: In some cases, alignment to ITU-T Recommendation X.694 [4] will force an order of elements in the mapped code (for instance using record instead of set) that is not originated neither in XML Schema nor in TTCN-3 requirements, but in ASN.1 specific considerations. In other cases, ASN.1-friendly identifiers will be generated for compatibility.

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 compilation of these 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) Mixed content is not supported.

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

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

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

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

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

6 Built-in data types

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 which defines the relation of XSD primitive types to TTCN-3 types. Whenever a new *simpleType* is defined, with the base type a built-in one, it will 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 WhiteSpace).
NOTE 4: With the exception of *decimal* which does support *totalDigits*.

6.1.1 Length

The 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 (8 bits) of binary data. For data types derived by *list*, *length* is measured in number of list items. A length-restricted XSD type is mapped to a corresponding length restricted type in TTCN-3.

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 facet *minLength* describes, how many units of length a value of the given simple type at least must have. It is mapped to a *length* restriction in TTCN-3 with a set lower boundary and an open upper boundary. Usage of the attribute *fixed* (see clause 7.1.5) has to 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 facet *maxLength* describes, how many units of length a value of the given simple type at most must have. It is mapped to a *length* restriction in TTCN-3 with a set upper boundary and a lower boundary of zero. Usage of the attribute *fixed* (see clause 7.1.5) has to 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 facet *pattern* describes a constraint in terms of a regular expression applied on a value space of a data type. For string based types this can be directly translated using the support in TTCN-3 for defining regular expression patterns on character sequences. It is not supported for numerical or boolean types. As the syntax of XSD regular patterns differs from the syntax of the TTCN-3 regular expressions, a mapping of the pattern expression has to be applied. The symbols `(`, `)`, `|`, `[`, `]`, `^`, `-` do not change and are translated directly. For the mapping of all other symbols refer to tables 3 and 4.

Table 3: Translation of special characters

XSD	TTCN-3
.	?
\s	[□\t\n\r]
\S	[^□\t\n\t]
\d	\d
\D	[^\d]
\w	\w
\W	[^\w]
\i	[\w\d:]
\I	[^\w\d:]
\c	[\w\d.\-_:]
\C	[^\w\d.\-_:]

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 are mapped to an escaped character in TTCN-3 or directly to the character (e.g. '.', and '+'). The double quote character must be mapped to an escaped double quote character. Character categories and blocks (like `\p{Lu}` or `\p{IsBasicLatin}`) are not supported. The correctness of the regular expression mappings themselves should be checked according to clause B.1.5 of ES 201 873-1 [1].

EXAMPLE:

```
<simpleType name="e6">
   <restriction base="string">
      <pattern value="(ahi|eho|cre|dve)@(f|F)okus"/>
   </restriction>
</simpleType>
```

Will be mapped to the following TTCN-3 expresion:

```
type XSD.String E6 (pattern "(ahi|eho|cre|dve)@(f|F)okus")
with {
variant "name as uncapitalized";
};
```

6.1.5 Enumeration

The facet *enumeration* constraints the value space to a specified set of values for a type.

An enumeration facet belonging to a simple type definition with a variety of atomic that is derived by restriction (directly or indirectly) from xsd:string shall be mapped to enumeration values of a TTCN-3 enumerated type (see clause 6.2.4 of ES 201 873-1 [1]) as specified in the three items below:

a) For each member of the value of the enumeration facet, an enumeration item that is an identifier (i.e. without associated integer value) shall be added to the enumerated type, 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 current documents to the corresponding member of the value of the enumeration facet.

c) The members of the value of the enumeration facet shall be mapped in ascending lexicographical order and any duplicate members shall be discarded.

An enumeration facet belonging to a simple type definition with a variety of atomic that is derived by restriction (directly or indirectly) from xsd:integer shall be mapped to enumeration values of a TTCN-3 enumerated type (see clause 6.2.4 of ES 201 873-1 [1]) as specified in the three items below.

a) For each member of the value of the enumeration facet, an enumeration item that is an enumeration identifier plus the associated integer value shall be added to the enumeration type, 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) 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 value of the enumeration facet 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 value of the enumeration. 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="10"/>
    </xsd:restriction>
</xsd:simpleType>

//Is mapped to the TTCN-3 type definition:
type enumerated Integer_0_5_10 {int0(0), int5(5), int10(10)}
    with { variant "name as uncapitalized" }
```

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="color">
    <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 is preserved for the codec using appropriate encoding instructions.

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 is set to "*collapse*", only the string type *normalizedString* (see clause 6.2.2 Normalized string), *token* (see clause 6.2.2 Normalized string), *language* (see clause 6.2.13 Language), *Name* (see clause 6.2.4) and *NCName*
(see clause 6.2.6 NCName) are allowed to specify this facet.

6.1.7 MinInclusive

The *minInclusive* facet is only valid for numerical types. It specifies the lowest bound for a number, including the boundary. This is mapped to a range restriction in TTCN-3 with a given lower boundary and the upper boundary of the base type (or *infinity* if not set*)*.

EXAMPLE:

Mapping of elements of type integer with *minInclusive* facet:

```
<simpleType name="e9">
   <restriction base="integer">
      <minInclusive value="-5"/>
   </restriction>
</simpleType>
```

Is mapped to:

```
type XSD.Integer E9 (-5 .. infinity)
with { variant "name as uncapitalized"};
```

6.1.8 MaxInclusive

The m*axInclusive* facet is only valid for numerical types. It specifies the upmost bound for a number, including the boundary. This is mapped to a range restriction in TTCN-3 with a given upper boundary and the lower boundary of the base type (-*infinity* if not set).

EXAMPLE:

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

```
<simpleType name="e10">
   <restriction base="positiveInteger">
      <maxInclusive value="100"/>
   </restriction>
</simpleType>
```

Is mapped to:

```
type XSD.PositiveInteger E10 (1 .. 100)
with {
variant "name as uncapitalized"};
```

6.1.9 MinExclusive

The mapping of *minExclusive* is very similar to *minInclusive* (see clause 6.1.7) only the given bound is not part of the range. A direct mapping of this is not possible in TTCN-3, as ranges are always including the given boundaries. To get around this, a value *delta* needs to be defined which is the smallest possible number handled by the TTCN-3 compiler for a given type (e.g. 1 for *integer* types and something very small for a *double*). The boundary is then modified by adding the *delta*.

EXAMPLE: Considering the mapping result of the example in clause 6.1.7 a translation with *minExclusive* facet would look like:

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

type XSD.Integer E9a (-4 .. infinity)
with {
variant "name as uncapitalized"};
```

(The original boundary of `-5` has been modified by the addition of a delta of `1`).

6.1.10 MaxExclusive

The mapping of *maxExclusive* is very similar to *maxInclusive* (see clause 6.1.8) only the given bound is not part of the range. A direct mapping of this is not possible in TTCN-3, as ranges are always including the given boundaries. To get around this a value *delta* needs to be defined which is the smallest possible number handled by the TTCN-3 compiler for a given type (e.g. 1 for *integer* types and something very small for a *double*). The boundary is then modified by subtracting the *delta*.

EXAMPLE: Considering the mapping result of the example in clause 6.2.6 NCName a translation with *maxExclusive* facet would look like:

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

Is mapped to:

```
type XSD.PositiveInteger E10a (1 .. 99)
with {
variant "name as uncapitalized";
};
```

(The original boundary of `100` has been modified by the subtraction of a delta of `1`).

6.1.11 Total digits

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

Will be translated to:

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

6.2 String types

XSD string types are generally converted to TTCN-3 as subtypes of *universal charstring* or *octetstring*. For an overview of the allowed facets please refer to table 2. Following are details on the mapping of all string types of XSD.

NOTE: 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is translated to TTCN-3 using a plain octetstring:

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

A translation has to be aware of the fact that XSD *hexBinary* allows for the usage of lowercase letters (*a*, *b*, *c*, *d*, *e*, and *f*) for specification of values. These need to be converted to upper case for TTCN-3.

It is not legal to specify patterns for *hexBinary* types.

6.2.11 Base 64 binary

The XSD *base64Binary* type is 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 mapped into TTCN-3 is:

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

};
```

EXAMPLE:

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

will be translated as:

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

and a value:

```
template E14 MyBase64BinaryTemplate:= '546974616E52756C6573'O
```

will be encoded as:

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

6.2.12 Any URI

The *anyURI* type is 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 is translated to TTCN-3 using the following pattern-restricted charstring:

```
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 is not translated to TTCN-3.

6.3 Integer types

XSD integer types are generally converted to TTCN-3 as subtypes of integer-based types. For an overview of the allowed facets please refer to table 2. Following are details on the mapping of all integer types of XSD.

6.3.1 Integer

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

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

6.3.2 Positive integer

The *positiveInteger* type is translated to TTCN-3 as a range-restricted *integer*.

EXAMPLE:

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

6.3.3 Non-positive integer

The *nonPositiveInteger* type is translated to TTCN-3 as a range-restricted *integer*.

EXAMPLE:

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

6.3.4 Negative integer

The *negativeInteger* type is translated to TTCN-3 as a range-restricted *integer*.

EXAMPLE:

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

6.3.5 Non-negative integer

The *nonNegativeInteger* type is translated to TTCN-3 as a range-restricted *integer*.

EXAMPLE:

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

6.3.6 Long

The *long* type is 64bit based and 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 and 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 and 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 and 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 and 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 and 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 and 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 and 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 please refer to table 2 in clause 6.1. Following are details on the mapping of all float types of XSD.

6.4.1 Decimal

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

```
type float  Decimal
  with { variant "XSD:decimal"};
```

6.4.2 Float

The *float* type is 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 is 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 are generally converted to TTCN-3 as pattern restricted subtypes of *charstring*. For an overview of the allowed facets please refer to table 2. Following are details on the mapping of all time types of XSD.

6.5.1 Duration

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

```
type charstring  Duration (pattern ")
  with { variant "XSD:duration"};
```

6.5.2 Date and time

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

```
type charstring  DateTime (pattern ")
  with { variant "XSD:dateTime"};
```

6.5.3 Time

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

```
type charstring  Time (pattern ")
  with { variant "XSD:time"};
```

6.5.4 Date

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

```
type charstring  Date (")
  with { variant "XSD:date"};
```

6.5.5 Gregorian year and month

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

```
type charstring  GYearMonth (pattern ")
  with { variant "XSD:gYearMonth"};
```

6.5.6 Gregorian year

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

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

6.5.7 Gregorian month and day

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

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

6.5.8 Gregorian day

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

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

6.5.9 Gregorian month

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

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

6.6 Sequence types

XSD sequence types are generally converted to TTCN-3 as a *record of* their respective base types. For an overview of the allowed facets please refer to table 2. Following are details on the mapping of all sequence types of XSD.

6.6.1 NMTOKENS

The *NMTOKENS* type is 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 *IDREFS* type is 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 *ENTITIES* type is mapped to TTCN-3 using a *record* construct of type *ENTITY*:

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

6.6.4 QName

The *QName* type is 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 results (the assumed target namespace is http://www.organization.org/).

EXAMPLE:

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

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


<?xml version="1.0" encoding="UTF-8"?>
<E14a xmlns="http://www.organization.org/">
<name xmlns:ns="http://www.otherorganization.org/">ns:someName</name>
<refId>10</refId>
</E14a>
```

6.7 Boolean type

The *boolean* type is mapped to TTCN-3 using as a *boolean*:

```
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 will be realized 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 anyType is the base type of all complex definitions and the anySimpleType.

The *anySimpleType* is 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 String attr,
record of String elem_list
} with {

  variant "XSD:anyType";
  variant "anyAttributes 'attr'";
  variant "anyElement 'elem_list'";

};
```

Without support for mixed content. See also clause 7.7 (anyAttributes, anyElement).

7 Mapping XSD components

After mapping the basic layer of XML Schema (i.e. the built-in types) a mapping of the structures has to follow. Every structure that may appear, globally or not, needs to 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 are encountered during mapping. It is not complete: special attributes that are only used by a single XSD component are described in the corresponding subclauses. Table 5 and table 6 show which attributes are needed to 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	✓						✓
minOccurs	✓						✓
ref	✓	✓					✓
abstract	✓			✓			
block	✓			✓			
default	✓	✓					
fixed	✓	✓					
form	✓	✓					
type	✓	✓					
mixed				✓		✓	

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 are mapped to TTCN-3 as simple type references, e.g. any component mapping to a type with name **`typeName`** and an attribute `id="`**`ID`**`"` should result in an additional TTCN-3 type declaration:

```
type Typename ID;
```

7.1.2 Ref

The *ref* attribute may reference an id or any global type (see clause 7.2).

If the attribute is referring to an *id* it is directly mapped as a simple type, e.g. a component with an attribute `ref="MyREF"` is translated to:

```
type MyREF  Typename;
```

In the case that **REF** references a global type the name of the global type has to be substituted, e.g.:

```
type  GlobalType— Typename;
```

7.1.3 Name

The attribute *name* holds the specified name for an XSD component. A component without this attribute is either defined anonymously or given by a reference (see clause 7.1.2). Names are directly mapped to TTCN-3 identifiers; please refer to clause 5.2 on constraints and properties of this conversion.

7.1.4 MinOccurs and MaxOccurs

The *minOccurs* and *maxOccurs* attributes provide for the number of times a XSD component can appear in a context. It is translated to a length restricted *record of* in TTCN-3.

EXAMPLE:

For example a XSD component with *minOccurs* and *maxOccurs* attributes would be translated as:

```
<complexType name="e15">
   <sequence minOccurs="5" maxOccurs="10">
      <element name="foo" type="integer"/>
      <element name="bar" type="float"/>
   </sequence>
</complexType>
```

Is translated to - a length-restricted *record of* record:

```
type record E15
{
 record length(5 .. 10) of record
  {
  XSD.Integer foo,
  XSD.Float  bar
  } sequence_list
}
with
{
  variant "name as uncapitalized ";
};
```

If only one boundary is given, the other boundary is established by the type or default value of the XSD structure. If a boundary value is *unbounded* the TTCN-3 keyword *infinity* has to be used. Also if *minOccurs* has a value of *0*, and *maxOccurs* has a value of 1, or is not present (defaulting to 1), a translator has to make sure that the resulting field 'sequence' is optional.

EXAMPLE:

```
<complexType name="e15a">

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

results in:

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

7.1.5 Default and Fixed

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

The 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.

As 'default' and 'fixed' have no equivalent in TTCN-3 space, they will be mapped into codec instructions.

EXAMPLE:

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

will be translated as:

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

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

7.1.6 Form

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

7.1.7 Type

The *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

Mixed content is not supported. All content has to be described by a schema.

7.1.9 Abstract

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

7.1.10 Block and final

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

7.2 Schema component

This is the root component of a XSD declaration. It is translated to the general structure of a TTCN-3 module containing all mapped types of the XSD schema. All direct children of *schema* are treated global and therefore need to be identifiable by *id* or *name*.

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.

EXAMPLE:

An example of a globally defined element:

```
<element name="e16" type="typename"/>
```

is translated to:

```
type typename E16
with {
variant "element";
variant "name as uncapitalized ";
};
```

Locally defined elements will be mapped to fields, refer to clause 7.6 about examples on this kind of mapping.

Among the possible attributes an *element* may possess are the special attributes *nillable* and *substitutionGroup*.

The *nillable* attribute, when set to true, gives the possibility of an element having the special value `"xsi:nil"` value in any XML data. –This will be implemented using an appropriate encoding extension:

```
<complexType name="e16a">
   <sequence>
      <element name="foo" type="integer"/>
      <element name="bar" type="string" nillable="true"/>
   </sequence>
</complexType>
```

Is translated into:

```
type record E16a
{
 XSD.Integer foo,
 record {
     XSD.String content optional
     }  bar
}
with
{
  variant "useNil 'bar'";
  variant "name as uncapitalized";
};
```

Which allows e.g. the following encoding:

```
template t_E16a:=
{
 foo:=3,
 bar:= {
      content:=omit
     }
}


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

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

For conversion of other attributes refer to clause 7.1.

7.4 Attribute components

Attributes define valid qualifiers for XML data and are used when defining complex types. Just like elements, attributes 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. Attributes are basically mapped in the same way as elements (see clause 7.3), an appended *with*- clause marking them as attributes.

EXAMPLE 1: For example, a globally defined attribute:

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

is mapped to:

```
type typename E17 with {

  variant "attribute";
  variant "name as uncapitalized ";
};
```

Locally defined attributes will be mapped to a TTCN-3 record containing a reference to the type definition they are belonging to or their assigned name. The generated structures are also appended with a with-clause marking the fields in question as attributes.

EXAMPLE 2: Take for an example a generated with-clause for a type with two attributes `foo` and `bar`:

```
<complexType name="e17a">
      <attribute name="foo" type="integer"/>
      <attribute name="bar" type="float"/>
</complexType>


type record E17a
{
   XSD.Float  bar optional,
      XSD.Integer foo optional
}
with
{
 variant "attribute 'foo', 'bar'";
variant "name as uncapitalized ";
};
```

While:

```
<attribute name="foo" type="string"/>

<complexType name="e17b">
  <sequence>
   <element name="e" type="string"/>
 </sequence>
 <attribute ref="foo"/>
 <attribute name="bar" type="string"/>
</complexType>


type XSD.String Foo with {
  variant "attribute";
  variant "name as uncapitalized ";
}

 type record E17b
{
  Foo    foo,
  XSD.String bar,
  XSD.String e

}
with
{
 variant "attribute 'foo', 'bar'";
 variant "name as uncapitalized ";
}
```

Refer to the appropriate subsections of clause 7.6 ComplexType components for examples on this kind of mapping.

Besides the general attributes (as laid out in clause 7.1 Attributes of XSD component declarations) attribute declarations may contain the special attribute *use*. The *use* attribute specifies whether an attribute (declared inside a structured type) is mandatory or not. The values of this attribute are: *optional*, *prohibited* and *required*. The value *required* does not to be translated as the existence of values is mandatory in TTCN-3. The value *prohibited* is used only in case of restricting *complexTypes* (see clauses 7.6.1.1 or 7.6.2.1 on restricting content of complex types). The value *optional* is translated by using the TTCN-3 keyword *optional* with the appropriate fields.

7.5 SimpleType components

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 Attributes of XSD component declarations. Please note that a `simpleType` is not allowed to contain elements or attributes, redefinition of these is done by using `complexType` (see clause 7.6 ComplexType components).

7.5.1 Derivation by restriction

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

It is also possible 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 is omitted and the type of the inner, anonymous simpleType is 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>
      <simpleType>
         <restriction base="string"/>
      </simpleType>
      <pattern value="(ahi|eho|cre|dve)@(f|F)okus"/>
   </restriction>
</simpleType>
```

This will generate a mapping for the inner type and a restriction thereof:

```
type XSD.String E18 (pattern "((ahi|eho|cre|dve)@(f|F)okus)#(1)")
with {
    variant "name as uncapitalized ";
};
```

7.5.2 Derivation by list

XSD list components are mapped to the TTCN-3 *record of* type. In their simplest form lists are 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 'E19' as uncapitalized";
};
```

When using any of the supported facets (length, maxLength, minLength) the translation is more complex and follows the mapping for built-in list types, with the difference that the base type is determined by an anonymous inner list item type (This is similar to clause **Error! Reference source not found.**).

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 facets are mapped accordingly (refer to respective 6.1 Mapping of facets clauses). If no *itemType* is given, the mapping has to be implemented using the given inner type (for an example refer to clause 7.5.3 Derivation by union).

7.5.3 Derivation by union

A union is considered as a set of mutually exclusive alternative types for a *simpleType*. As this is compatible with the *union* concept of TTCN-3, a *simpleType union* in XSD is mapped to a union structure in TTCN-3.

EXAMPLE 1:

```
<simpleType name="e21">
   <union>
      <simpleType>
         <restriction base="string"/>
      </simpleType>
      <simpleType>
         <restriction base="float"/>
      </simpleType>
   </union>
</simpleType>
```

Results in the following mapping:

```
type union E21
{
   XSD.String alt_0,
   XSD.Float alt_1
} with {
  variant "name 'alt_0' as ''";
  variant "name 'alt_1' as ''";
  variant "useUnion"
  variant "name as uncapitalized";
};
```

EXAMPLE 2:

For instance, the below structure:

```
type record E21 e21a
{
      E21 e21,
   XSD.String foo
}

template t_E21a:={
  e21:={
   alt_0:="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'>
<e21 xsi:type="string">ding</e21>
<foo>foostring</foo>
</e21a>
```

Using the attribute *memberTypes*, a union allows for a direct specification of member types as a whitespace separated list of type identifiers. The mapping to TTCN-3 is done in the same way as for the attribute *listItem* of the list derivation component (see clause 7.5.2Derivation by list).

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

EXAMPLE 3: Consider this example:

```
<simpleType name="e22">
   <restriction base="e21">
      <enumeration value="20"/>
      <enumeration value="50"/>
      <enumeration value="small"/>
   </restriction>
</simpleType>
```

Translates to:

```
type E21 E22 ({alt_1:=20.0},{alt_1:=50.0},{alt_0:="small"})
with {
  variant "name as uncapitalized";
};
```

7.6 ComplexType components

The *complexType* is used for creating new types that contain other elements and attributes. Just like *simpleType*, *complexType* may be defined globally (as child of *schema* or *redefine*). In this case the *name* attribute is mandatory and the resulting TTCN-3 type will be mapped to the value of this attribute. A *complexType* may also be defined locally (as a child of *element*) in an anonymous fashion (without the *name* attribute), therefore prohibiting to be referenced from other type definitions.

The mapping of a *complexType* is done by translating every child that this *complexType* may have and subsequently combining them by using a TTCN-3 record type or direct reference. A record type has to be used when the *complexType* contains attributes and a direct reference is used when no additional attributes are declared. The content of a *complexType* consists of either a *simpleContent* or *complexContent* component or a valid combination of *group, all, choice, sequence, attribute, attributeGroup* or *anyAttribute* components (see clause 3.4.2 in [8]). Following is a description of the mapping for the different possible content components.

7.6.1 ComplexType containing simple content

A *simpleContent* component is translated to types that may only have a *simpleType* as base. It is possible to extend or restrict the base type and to add attributes, but not elements.

7.6.1.1 Extending simple content

When extending *simpleContent* further attributes may be added to the original type. The example below extends a built-in type by adding an attribute. The mapping result of an extended *simpleContent* type with added attributes is always a record containing the base type as a field referenced by the reserved name *base*.

EXAMPLE:

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

Will be mapped as:

```
type record E23
{
      XSD.Float foo optional,
   XSD.String base
} with {
  variant "untagged 'base'";
  variant "attribute 'foo'";
  variant "name as uncapitalized";

};
```

7.6.1.2 Restricting simple content

To restrict *simpleContent* additional, more restrictive, facets are applied to the base type or to attributes of the base type. The whole type needs to be redefined in the restricted version, translating to a completely new type definition in TTCN-3.

EXAMPLE: Consider the following example for restriction of a base type:

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

Is translated to:

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

Other base types are dealt with accordingly, see clause 6.

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 by adding attributes or elements, it is also possible to restrict a base type to certain elements or attributes.

7.6.2.1 Extending complex content

By using extension for a *complexContent* it is possible to add attributes, elements or groups of those (*group*, *attributeGroup*) to the complex base type. This is translated to TTCN-3 by creating a record containing the referenced structure and the extension components.

EXAMPLE 1: For an example consider the following *complexType*:

```
<complexType name="e25">
   <sequence>
      <element name="title" type="string"/>
      <element name="forename" type="string"/>
      <element name="surname" type="string"/>
   </sequence>
</complexType>
```

The resulting mapping (according to clause 7.6.6 Sequence content) of the above *complexType* is:

```
type record E25
{

XSD.String title,
XSD.String forename,
XSD.String surname
}
with
{
  variant "name as uncapitalized ";
};
```

Now a type is defined that extends e25 by adding a new element:

```
<complexType name="e26">
   <complexContent>
      <extension base="e25">
         <sequence>
            <element name="age" type="integer"/>
         </sequence>
      </extension>
   </complexContent>
</complexType>
```

This is translated to the TTCN-3 structure:

```
type record E26
{

XSD.String title,
XSD.String forename,
XSD.String surname,
XSD.Integer age
}
with
{
  variant "name as uncapitalized ";
};
```

As the base content and extension content are of the same structure (*sequence*), the extension content is directly mapped to the result structure. In the case of different structures a nested structure would be generated.

EXAMPLE 2: Consider an extension with a different structure:

```
<complexType name="e27">
   <complexContent>
      <extension base="e25">
         <choice>
            <element name="age" type="integer"/>
            <element name="birthday" type="date"/>
         </choice>
      </extension>
   </complexContent>
</complexType>
```

This is translated to the following TTCN-3 structure:

```
type record E27
{
XSD.String title,
XSD.String forename,
XSD.String surname,
   union {
         XSD.Integer age,
         XSD.Date  birthday
            } choice
}
with
{
  variant "name 'E27' as uncapitalized ";
  variant "untagged 'choice'";
};Restricting complexContent
```

The *restriction* uses a base type and restricts some of its components. This is mapped to a new type containing only the components of the restriction. In the example below *anyType* (any possible type) is used as the base type and it is restricted to only two elements. As the resulting type is a *sequence* without additional attributes a TTCN-3 *record* is used (see clause 7.6.6 Sequence content on mapping of sequences), otherwise a *set* would be constructed containing base type and attributes.

EXAMPLE 3:

```
<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 ";
};
```

7.6.3 Group components

A *group* component defines an atomic group of elements for inclusion in other definitions. Groups can be local (anonymous) or global (with a mandatory *name* or *id*) and may consist of a single *choice*, *sequence* or *all* component. This translates to a single TTCN-3 type definition which is identified by the *name* (resp. *id*) of the *group*. Refer to the appropriate parts of clause 7.6 ComplexType components for more detailed information on the mapping of *choice*, *sequence* or *all* structures.

7.6.4 All content

An *all* content structure defines an unordered collection of optional elements. This is translated in TTCN-3 as a record with elements mapped to optional fields and a first field named 'order' reflecting the order and presence of fields within the record.

EXAMPLE:

```
<complexType name="e29">
   <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 E29
{
  record of enumerated {foo,bar,ding} order,
   XSD.Integer foo optional,
   XSD.Float  bar optional,
   XSD.String ding optional
}
with
{
  variant "name 'E29' as uncapitalized ";
  variant "useOrder";
};
```

NOTE: When encoding, the presence and order of elements in the encoded XML instance will be controlled by the 'order' field. 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. This mapping is required by the alignment to ITU-T Recommendation X.694 [4], originating in the PER encoders' possibility to reorder elements in a set.

7.6.5 Choice content

A *choice* content defines a collection of mutually exclusive alternatives for a type. It is thus mapped to the *union* type in TTCN-3, as it allows only one of the components to appear in the instance. The content for a choice component may be any combination of *element*, *group*, *choice*, *sequence* or *any*.

The following clauses give examples of the mapping for various contents nested in a choice component.

7.6.5.1 Choice with nested element

Nested elements are mapped to a union containing the choice's content.

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 "untagged 'choice'";
};
```

7.6.5.2 Choice with nested group

Nested group components will be mapped along other content as a field in the *union*.

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="e31"/>
      <element name="ding" type="string"/>
   </choice>
</complexType>
```

The *group* is mapped to a *record* (it is a *sequence*, see clause 7.6.6 Sequence content) and then the *choice* is translated to a TTCN-3 *union*:

```
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 "untagged 'choice'";
};
```

7.6.5.3 Choice with nested choice

A *choice* with a nested *choice* is translated as nested *union*s in TTCN-3.

EXAMPLE:

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

Will be mapped as:

```
type record E33
{
 union
 {
      union
      {
         XSD.String foo,
         XSD.String bar
      } choice,
      XSD.String ding
 }choice

} with
{
   variant "name as uncapitalized ";
   variant "untagged 'choice', 'choice.choice'";
 };
```

7.6.5.4 Choice with nested sequence

A *choice* with a nested *sequence* will be mapped to a *union* containing a *record*.

EXAMPLE:

```
<complexType name="e34">
   <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 E34
  {
   union
   {
        record
        {
           XSD.String foo,
           XSD.String bar
        }     sequence,
        XSD.String ding
   } choice

  } with
  {
    variant "name as uncapitalized ";
    variant "untagged 'choice','choice.sequence'";
  };
```

7.6.5.5 Choice with nested any

A *choice* containing XSD *any* types.

EXAMPLE:

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

Will translate to:

```
  type record E35
  {

  union {
  XSD.String foo,
  XSD.String elem
        } choice
  }
  with
  {
    variant "name as uncapitalized";
    variant "anyElement 'elem' from "other" ";
    variant "untagged 'choice'"
  };
```

See also clause 7.7 for details.

7.6.6 Sequence content

A sequence defines an ordered collection of components and is mapped to a record in TTCN-3. The content of a sequence may be any combination of element, group, choice, sequence or any.

The following clauses give examples of the mapping for various contents nested in a sequence component.

7.6.6.1 Sequence with nested element content

Sequences that contain only elements are mapped as a plain *record* in TTCN-3.

EXAMPLE 1: Mapping a sequence content:

```
  <complexType name="e36">
     <sequence>
        <element name="foo" type="integer"/>
```

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

Is mapped to

```
type record E36
{
XSD.Integer foo,
XSD.Float  bar
}
with
{
  variant "name as uncapitalized";
};
```

EXAMPLE 2: Mapping a recursive inner type:

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

Is mapped to:

```
type record X {
  XSD.String x,
  record {
    X y optional,
    XSD.String z
  } y optional
}
```

7.6.6.2 Sequence with nested group content

Nested group components will be mapped along other content as a field in the *record*.

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="e37"/>
      <element name="ding" type="string"/>
   </sequence>
</complexType>
```

The *group* is mapped to a *union* (it is a *choice*, see clause 7.6.5 Choice content) and then the *sequence* is translated to a *record*:

```
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

A *sequence* with a nested *choice* will be mapped to a *record* containing a *union*.

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 "untagged 'choice'";
};
```

7.6.6.4 Sequence with nested sequence content

A *sequence* with a nested *sequence*.

EXAMPLE:

```
<complexType name="e40">
   <sequence>
      <sequence>
         <element name="foo" type="string"/>
         <element name="bar" type="string"/>
      </sequence>
      <element name="ding" type="string"/>
   </sequence>
</complexType>
```

Will be mapped as:

```
type record E40
{
   XSD.String foo,
   XSD.String bar,
   XSD.String ding
} with
{
  variant "name as uncapitalized";
};
```

7.6.6.5 Sequence with nested any content

A *sequence* with nested *any*.

EXAMPLE:

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

Will translate to:

```
type record E41
{
XSD.String foo,
XSD.String elem
}
with
{
  variant "name as uncapitalized";
  variant "anyElement 'elem' ";
};
```

See also clause 7.7 for details.

7.6.7 AttributeGroup Components

An *attributeGroup* defines a group of attributes that can be included together inside other definitions. Attribute groups map to a *set* in TTCN-3 with all optional fields and a mandatory *with*-extension marking them as an *attributeGroup.* They are globally defined (as direct child of *schema* or *redefine* and requiring a *name* attribute) and locally referenced (requiring a *ref* attribute). If an attributeGroup is referenced within another attributeGroup all attributes referenced need to be merged in a single set.

EXAMPLE 1: Consider the following example:

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

<attributeGroup name="e43">
   <attributeGroup ref="e42"/>
   <attribute name="ding" type="string"/>
</attributeGroup>
```

Translates to TTCN-3 as:

```
type set E42
{
   XSD.Float foo optional,
   XSD.Float bar optional
} with {
  variant "name 'E42' as uncapitalized";
  variant "attributeGroup";
};

type set E43
{
   XSD.Float foo optional,
   XSD.Float bar optional,
   XSD.String ding optional
} with {
  variant "name as uncapitalized";
  variant "attributeGroup"
};
```

If *attributeGroup* components are referenced from a *complexType*, *restriction* or *extension*, a reference to the *attributeGroup* is generated and resolved by insertion in the mapped construct.

EXAMPLE 2:

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


type record E44
{
   XSD.String ding,
   XSD.Float foo optional,
   XSD.Float bar optional
} with {
  variant "name as uncapitalized";
  variant "attribute 'foo','bar'"
};
```

7.7 Any and anyAttribute

An *any* element specifies that any well-formed XML is permissible in a type's content model. The content of this XML will 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 a corresponding *anyAttribute* element which enables transparent (from the codec's point of view) attributes to appear in elements.

The codec will be controlled by an 'anyAttribute' and 'anyElement' instruction, complemented with 'from' and 'except' clauses specifying the comma-separated list of namespaces which are allowed or restricted to qualify the given attribute or element. An 'unqualified' clause in the list of allowed namespaces allows unqualified attributes or elements, respectively, while an 'unqualified' clause in the list of restricted namespaces forbids such attributes or elements.
For the following examples, the target namespace is assumed to be the following URI:
http://www.organization.org/ttcn/wildcard.

EXAMPLE 1:

The below complexType definitions:

```
<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.organization.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>
```

will be mapped as follows:

```
type record E45
{
record of XSD.String  attr
} with
{
  variant "name as uncapitalized";
  variant "anyAttributes 'attr'";
};

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

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

type record E45c
{
record of XSD.String  attr
} with
{
  variant "name as uncapitalized";
  variant "anyAttributes 'attr' from
unqualified,'http://www.organization.org/ttcn/attribute'";
};


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

EXAMPLE 2:

The below examples of a content model wildcard:

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

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

<xs:complexType name="e46c">
<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 'E46' as uncapitalized";
  variant "anyElement 'elem'";
};

type record E46a
{
```

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

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

7.8 Annotation

An *annotation* is used to include additional information in the XSD data. Annotations may appear in every component and will be mapped to a corresponding comment in TTCN-3. The comment should 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 is to 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 !
```

Annex A (normative): XSD.ttcn3

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 RecommendationX.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 {


    //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 "anyAttributes 'attr'";
      variant "anyElement 'elem_list'";
    };

   // 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  DateTime (pattern ") with {
    variant "XSD:dateTime"
};

type charstring  Time (pattern ") with {
    variant "XSD:time"
};

type charstring  Date (pattern ") with {
    variant "XSD:date"
};

type charstring  GYearMonth (pattern ") with {
    variant "XSD:gYearMonth"
};

type charstring  GYear (pattern ") with {
    variant "XSD:gYear"
};

type charstring  GMonthDay (pattern "--((0[1-9])|[10-12])-((0[1-9])|[10-31])([+-
]?[\d:Z]+)#(0,1)") with {
    variant "XSD:gMonthDay"
};

type charstring  GDay (pattern ") with {
    variant "XSD:gDay"
};

type charstring  GMonth (pattern "--((") 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 (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.

B.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 addressField,
   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 "attribute 'orderid'";
}


} 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>
```

B.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) "name as capitalized, attribute";
     variant(a2) "name as capitalized, attribute";
   variant "untagged 'base'"
   }
} with {
encode "XML";
}


module Example2Templates {

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

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

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

B.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) "attribute";
     variant(a2) "attribute";
   variant "untagged ‘base’";
   }



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

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

variant "namespace all as 'nsA' "
}


module Example3Templates {

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

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


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


   template 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>
```

B.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
   type Example2.S1

} with {
encode “XML”;
variant "name all as uncapitalized";
variant “namespace all 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 t_NewC1:= {
      base  :=-1000,
      a1    :=1,
      a2    :=2
   }

   template 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>
```

History

Document history		
V3.2.2	May 2008	Membership Approval Procedure MV 20080704: 2008-05-06 to 2008-07-04
V3.3.1	July 2008	Publication