ASN1C Mapping of ASN.1 Syntax to XML Schema

Objective Systems, Inc., Last Update: July 2005

Table of Contents

Abstract

An effort is currently underway within the ITU-T to map World-Wide Web Consortium (W3C) XML Schema Definitions Language (XSD) to Abstract Syntax Notation 1 (ASN.1). But a parallel effort to provide a mapping in the other direction – from ASN.1 to XSD – is on hold. Given the currently popularity of XSD for defining new standards, it would seem reasonable that ASN.1 to XSD conversion would be of interest to the ASN.1 community.

This paper presents such a mapping. It uses as a basis the T1 Standard's Committee Draft Standard – *tML Guidelines for mapping ASN.1 syntax and modules to XML Schemas.*[1]

General Conventions

In this document, sections of ASN.1 syntax and XML Schema definitions are shown in plain text (`courier` font). Within these sections, symbols shown in *italics* indicate placeholders for items to be substituted. For example, in the statement *TypeName* ::= A, *TypeName* would be replaced with a valid ASN.1 name for a type.

This paper covers the conversion of ASN.1 module headers and types.

Mapping of Top-Level Constructs

An ASN.1 module name is mapped to an XML schema namespace. ASN.1 IMPORT statements are mapped to XSD *import* statements. The ASN.1 EXPORT statement does not have a corresponding construct in XSD.

The general form of the XSD namespace and import statements would be as follows:

```
<?xml version="1.0"?>
<xsd:schema
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   targetNamespace="URL/ModuleName"

   <!-- following line would be added for imported module namespace -->
   xmlns:ImportedModuleName="importURL/ImportedModuleName"
      elementFormDefault="qualified">

   <xsd:import namespace="importURL/ImportedModuleName"
      schemaLocation="ImportedModuleName.xsd"/>
```

In this definition, the items in *italics* would be replaced with text from the ASN.1 specification being converted or a configuration file. The *ModuleName* and *ImportedModuleName* items would come from the ASN.1 specification. The *URL* and *importURL* items would be configuration parameters.

Mapping of ASN.1 Types

Each ASN.1 type is mapped to a corresponding XSD type. The following sections describe the mappings for each of the ASN.1 built-in types.

BOOLEAN

The ASN.1 BOOLEAN type is mapped to the XSD *boolean* built-in type.

ASN.1 production:

```
TypeName ::= BOOLEAN
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="xsd:boolean"/>
</xsd:simpleType>
```

INTEGER

The ASN.1 INTEGER type is converted into one of several XSD built-in types depending on value range constraints on the integer type definition.

The default conversion if the INTEGER value contains no constraints is to the XSD *integer* type:

ASN.1 production:

```
TypeName ::= INTEGER
```

Generated XSD code:

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

If the integer has a value range constraint that allows a more restrictive XSD type to be used, then that type will be used. For example, if a range of 0 to 255 (inclusive) is specified, an XSD *unsignedByte* would be used because it maps exactly to this range. The following table shows the range values for each of the INTEGER type mappings:

Lower Bound	Upper Bound	XSD Type
-128	127	byte
0	255	unsignedByte
-32768	32767	short
0	65535	unsignedShort
-2147483648	2147483647	integer
0	4294967295	unsignedInt
-9223372036854775808	9223372036854775807	long
0	18446744073709551615	unsignedLong

Ranges beyond "long" or "unsignedLong" will cause the integer value to be treated as a "big integer". This will map to an *xsd:string* type. An integer can also be specified to be a big integer using the ASN1C <isBigInteger/> configuration file setting.

If constraints are present on the INTEGER type that are not exactly equal to the lower and upper bounds specified above, then *xsd:minInclusive* and *xsd:maxInclusive* facets will be added to the XSD type mapping. For example, the mapping of "I ::= INTEGER (0..10)" would be done as follows:

1. The most restrictive type would first be chosen based on the constraints. In this case, *xsd:byte* would be used because it appears first on the list above.

2. Then the *xsd:minInclusive* and *xsd:maxInclusive* facets would be added to further restrict the type.

This would result in the following mapping:

```
<xsd:simpleType name="I">
   <xsd:restriction base="xsd:byte">
      <xsd:minInclusive value="0">
      <xsd:maxInclusive value="10">
   </xsd:restriction>
</xsd:simpleType>
```

BIT STRING

There is no built-in XSD type that corresponds to the ASN.1 BIT STRING type. For this reason, a custom type was created in the Objective Systems XSD run-time library (*asn1.xsd*) to model this type. This type is *asn1:BitString* and has the following definition:

```
<xsd:simpleType name="BitString">
   <xsd:restriction base="xsd:token">
      <xsd:pattern value="[0-1]{0,}"/>
   </xsd:restriction>
</xsd:simpleType>
```

The ASN.1 BIT STRING type is converted into a reference to this custom type as follows:

ASN.1 production:

```
TypeName ::= BIT STRING
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="asn1:BitString"/>
</xsd:simpleType>
```

Sized BIT STRING

The ASN.1 BIT STRING type may contain a size constraint. This is converted into *minLength* and *maxLength* facets in the generated XSD definition:

ASN.1 production:

```
TypeName ::= BIT STRING (SIZE (lower..upper))
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="asn1:BitString">
      <xsd:minLength value="lower"/>
      <xsd:maxLength value="upper"/>
   </xsd:restriction>
</xsd:simpleType>
```

BIT STRING with Named Bits

A bit string with named bits is handled differently than a normal bit string. This is because the primary use of named bits it to define a bit map of selected bit items. For this reason, a list of enumerated items is used for the type. This allows the names of the bits to be specified in an XML instance of the type. The type also contains application information in the form of an non-native attribute that allows an application to map the items specified in a list to binary bits in a bitmap.

The formal mapping of an ASN.1 BIT STRING with named bits to XSD is as follows:

ASN.1 production:

```
TypeName ::= BIT STRING { b1(n1), b2(n2) }
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
  <xsd:union memberTypes="asn1:BitString">
    <xsd:list>
      <xsd:simpleType>
        <xsd:restriction base="xsd:token">
          <xsd:enumeration value="b1" asn1:bitno="n1"/>
          <xsd:enumeration value="b2" asn1:bitno="n2"/>
          ...
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:list>
  </xsd:union>
</xsd:simpleType>
```

The previous version of this tool also generates the annotation code for this definition. This can also be generated with current version using the -appinfo option.

Generated XSD code with annotation:

```
<xsd:simpleType name="TypeName">
     <xsd:annotation>
       <xsd:appinfo>
         <asn1:NamedBitInfo>
           <asn1:NamedBit name="b1" bitNumber="n1">
           <asn1:NamedBit name="b2" bitNumber="n2">
           ...
         </asn1:NamedBitInfo>
       </xsd:appinfo>
     </xsd:annotation>
     <xsd:union memberTypes="asn1:BitString">
       <xsd:list>
         <xsd:simpleType>
           <xsd:restriction base="xsd:token">
             <xsd:enumeration value="b1"/>
             <xsd:enumeration value="b2"/>
             ...
           </xsd:restriction>
         </xsd:simpleType>
       </xsd:list>
     </xsd:union>
</xsd:simpleType>
```

The appinfo section will not be needed if the bits are sequentially numbered starting at zero. An application that uses the mapping would be able to calculate the bit numbers based on their position in the document.

Example

The following ASN.1 BIT STRING type:

```
ColorSet ::= BIT STRING (blue(1), green(3), red(5))
```

maps to the following XSD type:

```
<xsd:simpleType name="ColorSet">
  <xsd:union memberTypes="asn1:BitString">
    <xsd:list>
      <xsd:simpleType>
        <xsd:restriction base="xsd:token">
          <xsd:enumeration value="blue" asn1:bitno="1"/>
          <xsd:enumeration value="green" asn1:bitno="3"/>
          <xsd:enumeration value="red" asn1:bitno="5"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:list>
  </xsd:union>
</xsd:simpleType>
```

OCTET STRING

The ASN.1 OCTET STRING type is converted into the XSD *hexBinary* type.

ASN.1 production:

```
TypeName ::= OCTET STRING
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="xsd:hexBinary"/>
</xsd:simpleType>
```

Sized OCTET STRING

The ASN.1 OCTET STRING type may contain a size constraint. This is converted into *minLength* and *maxLength* facets in the generated XSD definition:

ASN.1 production:

```
TypeName ::= OCTET STRING (SIZE (lower..upper))
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="hexBinary">
      <xsd:minLength value="lower"/>
      <xsd:maxLength value="upper"/>
   </xsd:restriction>
</xsd:simpleType>
```

Character String Types

All ASN.1 character string useful types (IA5String, VisibleString, etc.) are mapped to the XSD *string* type.

ASN.1 production:

```
TypeName ::= ASN1CharStringType
```

in this definition, *ASN1CharStringType* would be replaced with one of the ASN.1 Character String types such as VisibleString.

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="string"/>
</xsd:simpleType>
```

ASN.1 character string types may contain a size constraint. This is converted into *minLength* and *maxLength* facets in the generated XSD definition:

ASN.1 production:

```
TypeName ::= ASN1CharStringType (SIZE (lower..upper))
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="xsd:string">
      <xsd:minLength value="lower"/>
      <xsd:maxLength value="upper"/>
   </xsd:restriction>
</xsd:simpleType>
```

ASN.1 character string types may also contain permitted alphabet or pattern constraints. These are converted into *pattern* facets in the generated XSD definition:

ASN.1 production:

```
TypeName ::= ASN1CharStringType (FROM (charSet))
```

or

```
TypeName ::= ASN1CharStringType (PATTERN (pattern))
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="xsd:string">
      <xsd:pattern value="pattern"/>
   </xsd:restriction>
</xsd:simpleType>
```

In this case, the permitted alphabet character set (*charSet*) is converted into a corresponding *pattern* for use in the generated XML schema definition.

Time String Types

The ASN.1 *GeneralizedTime* and *UTCTime* types are mapped to the XSD *dateTime* type.

ASN.1 production:

```
TypeName ::= ASN1TimeStringType
```

in this definition, *ASN1TimeStringType* would be replaced with either *GeneralizedTime* or *UTCTime*.

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="dateTime"/>
</xsd:simpleType>
```

ENUMERATED

The ASN.1 ENUMERATED type is converted into an XSD token type with enumeration items. The enumeration items correspond to the enumerated identifiers in the type. If the enumerated items contain numbers (i.e do not follow the standards sequence), then an <appinfo> annotation is added to the type to allow an application to map the enumerated identifiers to numbers. If <appinfo> is not present, then an application can safely assume that the enumerated identifiers are in sequential order starting at zero.

ASN.1 production:

```
TypeName ::= ENUMERATED (id1(val1), id2(val2), ...)
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="xsd:token">
      <xsd:enumeration name="id1" asn1:value="val1">
      <xsd:enumeration name="id2" asn1:value="val2">
   </xsd:restriction>
</xsd:simpleType>
```

The previous version of this tool also generates the annotation code for this definition. This can also be generated with current version using -appinfo option.

Generated XSD code with annotation:

```
<xsd:simpleType name="TypeName">
   <xsd:annotation>
      <xsd:appinfo>
         <asn1:EnumInfo>
           <asn1:EnumItem name="id1" value="val1"/>
           <asn1:EnumItem name="id2" value="val2"/>
         </asn1:EnumInfo>
      </xsd:appinfo>
   </xsd:annotation>
   <xsd:restriction base="xsd:token">
      <xsd:enumeration name="id1">
      <xsd:enumeration name="id2">
   </xsd:restriction>
</xsd:simpleType>
```

Example

The following ASN.1 enumerated type:

```
Colors ::= ENUMERATED (blue(1), green(3), red(5))
```

maps to the following XSD type

```
<xsd:simpleType name=”Colors”>
   <xsd:restriction base="xsd:token">
      <xsd:enumeration name=”blue” asn1:value=”1”>
      <xsd:enumeration name=”green” asn1:value=”3”>
      <xsd:enumeration name=”red” asn1:value=”5”>
   </xsd:restriction>
</xsd:simpleType>
```

Note that if the identifiers in the enumerated type did not contain numbers (i.e. if the type was ‘ENUMERATED (blue, green, red)’), then the annotation would not be necessary on the type above.

NULL

There is no built-in XSD type that corresponds to the ASN.1 NULL type. For this reason, a custom type was created in the Objective Systems XSD run-time library (*asn1.xsd*) to model this type. This type is *asn1:NULL* and has the following definition:

```
<complexType name=”NULL” final=”#all”/>
```

This is a non-extendable empty complex type.

OBJECT IDENTIFIER

There is no built-in XSD type that corresponds to the ASN.1 OBJECT IDENTIFIER type. For this reason, a custom type was created in the Objective Systems XSD run-time library (*asn1.xsd*) to model this type. This type is *asn1:ObjectIdentifier* and has the following definition:

```
<xsd:simpleType name="ObjectIdentifier">
   <xsd:restriction base="xsd:token">
      <xsd:pattern value=
         ”[0-2](((\.[1-3]?[0-9])(\.\d+)*)?”/>
   </xsd:restriction>
</xsd:simpleType>
```

The pattern enforces the rule in the X.680 standard that the first arc value of an OID must be between 0 and 2, the second arc must be between 0 and 39, and the remaining arcs can be any number. The ASN.1 OBJECT IDENTIFIER type is converted into a reference to this custom type as follows:

ASN.1 production:

```
TypeName ::= OBJECT IDENTIFIER
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
```

```
      <xsd:restriction base="asn1:ObjectIdentifier"/>
   </xsd:simpleType>
```

RELATIVE-OID

There is no built-in XSD type that corresponds to the ASN.1 RELATIVE-OID type. For this reason, a custom type was created in the Objective Systems XSD run-time library (*asn1.xsd*) to model this type. This type is *asn1:RelativeOID* and has the following definition:

```
<xsd:simpleType name="RelativeOID">
   <xsd:restriction base="xsd:token">
      <xsd:pattern value="\d+(\.\d+)*"/>
   </xsd:restriction>
</xsd:simpleType>
```

This is similar to the OBJECT IDENTIFIER type discussed in the previous section except in this case, the patter in simpler. The arc numbers in a RELATIVE-OID are not restricted in any way, hence the simpler pattern. The ASN.1 RELATIVE-OID type is converted into a reference to this custom type as follows:

ASN.1 production:

```
TypeName ::= RELATIVE-OID
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="asn1:RelativeOID"/>
</xsd:simpleType>
```

REAL

The ASN.1 REAL type is mapped to the XSD *double* built-in type.

ASN.1 production:

```
TypeName ::= REAL
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="xsd:double"/>
</xsd:simpleType>
```

SEQUENCE

An ASN.1 SEQUENCE is a constructed type consisting of a series of element definitions that must appear in the specified order. This is very similar to the XSD *sequence* complex type and is therefore mapped to this type.

The basic mapping is as follows:

ASN.1 production:

```
TypeName ::= SEQUENCE {
   element1-name element1-type,
   element2-name element2-type,
      ...
   }
```

Generated XSD code:

```
<xsd:complexType name="TypeName">
   <xsd:sequence>
      <xsd:element name="element1-name" type="element1-type"/>
      <xsd:element name="element2-name" type="element2-type"/>
      …
   </xsd:sequence>
</xsd:complexType>
```

OPTIONAL keyword

Elements within a sequence can be declared to be optional using the OPTIONAL keyword. This indicates that the element is not required in the encoded message. XSD contains the *minOccurs* facet that can be used to model this behavior. Setting *minOccurs* equal to zero is the equivalent to declaring an element to be optional because this indicates the element can appear zero to one times in the definition.

For example, the following ASN.1 SEQUENCE type:

```
OptInt ::= SEQUENCE {
   anInt    INTEGER OPTIONAL
}
```

will cause the following XSD complex type to be generated:

```
<xsd:complexType name="OptInt">
  <xsd:sequence>
    <xsd:element name="anInt" type="xsd:integer" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

DEFAULT keyword

The DEFAULT keyword allows a default value to be specified for elements within the SEQUENCE. XSD contains a *default* facet that can be used to map elements with this keyword.

For example, the following ASN.1 SEQUENCE type:

```
DfltInt ::= SEQUENCE {
   anInt    INTEGER DEFAULT 1
}
```

will cause the following XSD complex type to be generated:

```
<xsd:complexType name="DfltInt">
  <xsd:sequence>
    <xsd:element name="anInt" type="xsd:integer" default="1"/>
```

```
      </xsd:sequence>
    </xsd:complexType>
```

Note that in XSD, default values can only be specified for simple (primitive) types. ASN.1 allows for the specification of default values on complex (constructed) types as well. If an ASN.1 type is encountered that contains a complex default value, the value is dropped in the conversion to XSD.

Extension Elements

If the SEQUENCE type is extensible (i.e., contains an ellipses marker …), a special element will be inserted to allow unknown elements to be validated. This special element is as follows:

```
<xsd:any namespace="##other" processContents="lax"/>
```

This element declaration allows any additional elements from other namespaces to exist in a message instance without causing a validation or decoding error. Note the restriction that the element must be defined in a different namespace. This is necessary because if the element existed in the same namespace as other elements, the content model would be non-deterministic. The reason is because a validation program would not be able to determine if the last element is a sequence was a defined element or an extension element.

The extension element is marked with a non-native attribute description. For example:

```
DfltInt ::= SEQUENCE {
   anInt   INTEGER,
   ...,
   extElm  BOOLEAN
}
```

Generated XSD code:

```
   <xsd:complexType name="DfltInt">
      <xsd:sequence>
         <xsd:element name="anInt" type="xsd:integer"/>
         <xsd:element name="extElm" minOccurs="0" type="xsd:boolean"
            asn1:description="extension element"/>
         <xsd:any namespace="##other" processContents="lax"
            minOccurs="0"/>
      </xsd:sequence>
   </xsd:complexType>
```

Note: In the ASN.1 SEQUENCE or SET type, all extension elements are optional, whether marked that way or not.

SET

An ASN.1 SET is a constructed type consisting of a series of element definitions that can appear in any order. This is similar to the XSD *all* complex type and is therefore mapped to this type.

The basic mapping is as follows:

ASN.1 production:

```
   TypeName ::= SET {
      element1-name element1-type,
```

```
    element2-name element2-type,
       ...
    }
```

Generated XSD code:

```
<xsd:complexType name="TypeName">
   <xsd:all>
      <xsd:element name="element1-name" type="element1-type"/>
      <xsd:element name="element2-name" type="element2-name"/>
      …
   </xsd:all>
</xsd:complexType>typedef struct {
```

The rules for mapping elements with optional and default values to XSD that were described in the SEQUENCE section above are also applicable to the SET type.

SEQUENCE OF / SET OF

The ASN.1 SEQUENCE OF or SET OF type is used to specify a repeating collection of a given element type. This is similar to an array type in a high-level programming language. For all practical purposes, SEQUENCE OF and SET OF are identical. The remainder of this section will refer to the SEQUENCE OF type only. It can be assumed that all of the defined mappings apply to the SET OF type as well.

The way the SEQUENCE OF type is mapped to XSD depends on the type of the referenced element. If the type is one of the following ASN.1 primitive types (or a type reference that references one of these types):

- BOOLEAN
- INTEGER
- ENUMERATED
- REAL

The mapping is to the XSD *list* type. This is a list of space-separated identifiers. The syntax is as follows:

ASN.1 production:

```
TypeName ::= SEQUENCE OF ElementType
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:list itemType="ElementType">
</xsd:simpleType>
```

This will be referred to as the simple case from this point forward.

If the element type is any other type than those listed above, the ASN.1 type is mapped to an XSD sequence complex type that contains a single element of the element type. The generated XSD type also contains the *maxOccurs* (and possibly the *minOccurs*) facet to specify the array bounds.

The general mapping of an unbounded SEQUENCE OF type (i.e. one with no size constraint) to XSD is as follows:

ASN.1 production:

```
TypeName ::= SEQUENCE OF ElementType
```

Generated XSD code:

```
<xsd:complexType name="TypeName">
   <xsd:sequence>
      <xsd:element name="ElementType" type="ElementType"
       maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
```

In this definition, the element type name is the name of the ASN.1 element type. The element type in the XSD definition is the equivalent XSD type for the ASN.1 element type.

As of the 2002 version of the ASN.1 standards, it is now possible to include an element identifier name before the element type name in a SEQUENCE OF definition. This makes it possible to control the name of the element used in the generated XSD definition. The mapping for this case is as follows:

ASN.1 production:

```
TypeName ::= SEQUENCE OF elementName ElementType
```

Generated XSD code:

```
<xsd:complexType name="TypeName">
   <xsd:sequence>
      <xsd:element name="elementName" type="ElementType"
       maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
```

Example

The following shows the mapping for a SEQUENCE OF INTEGER. Since INTEGER is one of the simple types listed above, an XSD *list* type is used:

ASN.1 production:

```
SeqOfInt ::= SEQUENCE OF INTEGER
```

Generated XSD code:

```
<xsd:complexType name="SeqOfInt">
   <xsd:list itemType="xsd:integer">
</xsd:complexType>
```

The following shows the mapping for a SEQUENCE OF UTF8String. Since UTF8String is not one of the simple types listed above, an XSD *sequence* type is used:

ASN.1 production:

```
SeqOfUTF8 ::= SEQUENCE OF UTF8String
```

Generated XSD code:

```
<xsd:complexType name="SeqOfUTF8">
   <xsd:sequence>
      <xsd:element name="UTF8String" type="xsd:string"
       maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
```

Note that the element name is the name of the element type. To change the element name, the ASN.1 form that allows an element name could be used:

ASN.1 production:

```
SeqOfUTF8 ::= SEQUENCE OF myString UTF8String
```

Generated XSD code:

```
<xsd:complexType name="SeqOfUTF8">
   <xsd:sequence>
      <xsd:element name="myString" type="xsd:string"
       maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
```

Sized SEQUENCE OF / SET OF

The SEQUENCE OF type may contain a size constraint. If this is the case, the XSD *minOccurs* and *maxOccurs* facets are used to constrain the value to the given size.

ASN.1 production:

```
TypeName ::= SEQUENCE (SIZE (lower..upper)) OF ElementType
```

Generated XSD code:

```
<xsd:complexType name="TypeName">
   <xsd:sequence minOccurs="lower" maxOccurs="upper">
      <xsd:element name="ElementType" type="ElementType"/>
   </xsd:sequence>
</xsd:complexType>
```

This mapping is for the complex case. For the simple case (i.e XSD *list* case), the XSD *minLength* and/or *maxLength* facets are used to constraint the length:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction>
      <xsd:simpleType>
         <xsd:list itemType="ElementType">
      </xsd:simpleType>
      <xsd:minLength value="lower"/>
      <xsd:maxLength value="upper"/>
   </xsd:restriction>
</xsd:simpleType>
```

CHOICE

The ASN.1 CHOICE type is used to specify a list of alternative elements from which a single element can be selected. This type is mapped to the XSD *choice* complex type. The mapping is as follows:

ASN.1 production:

```
TypeName ::= CHOICE {
   element1-name element1-type,
   element2-name element2-type,
      ...
   }
```

Generated XSD code:

```
<xsd:complexType name="TypeName">
   <xsd:choice>
      <xsd:element name="element1-name" type="element1-type"/>
      <xsd:element name="element2-name" type="element2-name"/>
      …
   </xsd:choice>
</xsd:complexType>
```

This is similar to the SEQUENCE and SET cases described above. The only difference is that *xsd:choice* is used instead of *xsd:sequence* or *xsd:all*.

The CHOICE type cannot have elements marked as optional (OPTIONAL) or elements that contain default values (DEFAULT) as was the case for SEQUENCE and SET.

Open Type

An *Open Type* as defined in the X.680 standard is specified as a reference to a *Type Field* in an *Information Object Class*. The most common form of this is when the *Type* field in the built-in TYPE-IDENTIFIER class is referenced as follows:

```
TYPE-IDENTIFIER.&Type
```

See the section in this document on Information Objects for a more detailed explanation.

There is no built-in XSD type that corresponds to the ASN.1 Open or ANY type. For this reason, a custom type was created in the Objective Systems XSD run-time library (*asn1.xsd*) to model this type. This type is *asn1:OpenType* and has the following definition:

```
<xsd:complexType name="OpenType">
   <xsd:sequence>
      <xsd:any processContents="lax" minOccurs="1" />
   </xsd:sequence>
</xsd:complexType>
```

The ASN.1 Open or ANY type is converted into a reference to this custom type as follows:

ASN.1 production:

```
TypeName ::= TYPE-IDENTIFIER.&Type

     Or

TypeName ::= ANY
```

Generated XSD code:

```
<xsd:complexType name="TypeName">
   <xsd:sequence>
      <xsd:any processContents="lax" minOccurs="1" />
   </xsd:sequence>
</xsd:complexType>
```

An example showing how an open type might be referenced in a SEQUENCE type and the corresponding conversion to XSD is as follows:

```
SeqWithOpenType ::= SEQUENCE {
   anOpenType     TYPE-IDENTIFIER.&Type
}
```

Generated XSD type:

```
<xsd:complexType name="SeqWithOpenType">
  <xsd:sequence>
    <xsd:element name="anOpenType" type="asn1:OpenType/>
  </xsd:sequence>
</xsd:complexType>
```

In this case, any valid XML instance can be used for the element.

Tagged Types

In ASN.1, it is possible to create new custom type using ASN.1 tag values as identifiers. These identifiers are built into BER or DER encoded messages. In general, these tags have no meaning in an XSD representation of an ASN.1 type that is used to create or validate XML markup. However, if the schema definition is to be used to generate a BER or DER instance of a type, the tag information will be required. For this reason, it is possible to add an application information annotation (*appinfo*) to the generated XSD type.

The annotation carries all of the information an application would need to know to encode a BER or DER message of the given type. This includes the tag's class, identifier number, and how it is applied (IMPLICIT or EXPLICIT). The type that specifies this information is the *TagInfo* type in the Objective Systems' XSD class library.

The mapping of an ASN.1 tagged type to XSD is as follows:

ASN.1 production:

```
TypeName ::= Tagging [ TagClass TagID ] ASN1Type
```

Generated XSD code:

```
<xsd:complexType name="TypeName" asn1:tag="[TagClass TagID]"
```

```
   asn1:tagging="EXPLICIT">
   equivalent XSD type mapping for ASN1Type
</xsd:complexType>
```

The previous version of this tool generates an annotation for tag information. This can also be generated with current version using the *-appinfo* option.

Generated XSD code with annotation:

```
<xsd:complexType name="TypeName">
   <xsd:annotation/>
      <xsd:appinfo>
         <asn1:TagInfo>
            <asn1:TagClass> TagClass </asn1:TagClass>
            <asn1:TagID> TagID </asn1:TagID>
            <asn1:Tagging> Tagging </asn1:Tagging>
         </asn1:TagInfo>
      </xsd:appinfo>
   </xsd:annotation>
   equivalent XSD type mapping for ASN1Type
</xsd:complexType>
```

Tagging in the definition above is optional. If present, it is equal to either the keyword EXPLICIT or IMPLICIT. The default value is EXPLICIT. A default value for all types in a module can also be specified in the ASN.1 module header.

The tag's form (constructed or primitive) is not specified in the mapping above. This is because this can be determined by an application that is encoding or decoding a message of the given type.

EXTERNAL and EmbeddedPDV Types

The EXTERNAL and EmbeddedPDV types are built-in ASN.1 types that make it possible to transfer a value of a different encoding type within an ASN.1 message. These are constructed types built into the ASN.1 standard. An XSD representation of each of these types is available in the *asn1.xsd* library. The ASN1C compiler generates a reference to the types in the library when it encounters a reference to one of these types.

ASN.1 production:

```
TypeName ::= EXTERNAL
```

Generated XSD code:

```
<xsd:complexType name="TypeName">
   <xsd:complexContent>
      <xsd:extension base="asn1:EXTERNAL"/>
   </xsd:complexContent>
</xsd:complexType>
```

ASN.1 production:

```
TypeName ::= EMBEDDED PDV
```

Generated XSD code:

```
<xsd:complexType name=”TypeName">
   <xsd:complexContent>
      <xsd:extension base=”asn1:EmbeddedPDV"/>
   </xsd:complexContent>
</xsd:complexType>
```

Mapping of ASN.1 Information Objects

The ITU-T ASN.1 X.681 and X.682 standards specify a table-drive approach for the assignment of constrained values to open types within a specification. These constraints are known as “table constraints” and utilize Open Type, Class, Information Object and ObjectSet definitions. A mapping is presented below for definitions of this type. This mapping will be generated by the ASN1C compiler or ASN2XSD translation tool if the *–tables* option is specified.

CLASS

An ASN.1 CLASS is used to define the structure of Information Objects and Information Object Sets. An Information Object Set is similar in structure to a relational table in that it is a collection of rows that define the set of messages that may be used in a constrained open type field. For this reason, CLASS is modeled as an unbounded collection of the fields defined within the CLASS definition.

The basic mapping is as follows:

ASN.1 definition:

```
ClassName ::= CLASS {
   Class field definitions…
}
```

Generated XSD code:

```
<xsd:complexType name=”ClassName">
   <xsd:sequence minOccurs=”0” maxOccurs=”unbounded”>
      <xsd:element name=”object”>
         <xsd:complexType>
            .. attribute mappings for class fields ..
         </xsd:complexType>
      </xsd:element>
   </xsd:sequence>
</xsd:complexType>
```

The only types of fields within a CLASS definition that results in an ASN.1-to-XSD mapping are type fields and fixed type value fields. These are translated to attributes with type *xsd:string*. Only simple value types are supported (i.e. those that have a direct mapping to a string) in this mapping. Fields of any other type (for example, object set fields) are ignored.

As an example, the mapping of a common 3GPP ASN.1 CLASS would be as follows:

ASN.1 definition:

```
NBAP-PROTOCOL-IES ::= CLASS {
```

```
    &id           ProtocolIE-ID UNIQUE,
    &criticality  Criticality,
    &Value        ,
    &presence     Presence}
  WITH SYNTAX {ID &id
               CRITICALITY &criticality
               TYPE &Value
               PRESENCE &presence}
```

Generated XSD code:

```
<xsd:complexType name="NBAP_PROTOCOL_IES">
   <xsd:sequence minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="object">
         <xsd:complexType>
            <xsd:attribute name="id" type="xsd:string"/>
            <xsd:attribute name="criticality" type="xsd:string"/>
            <xsd:attribute name="type" type="xsd:string"/>
            <xsd:attribute name="presence" type="xsd:string"/>
         </xsd:complexType>
      </xsd:element>
   </xsd:sequence>
</xsd:complexType>
```

In this definition, the following fields are fixed type value fields: id, criticality, and presence. Value is a type field (the attribute name 'type' is always used for type fields). All were translated to attributes within the inner complexType.

Information Object and Information Object Set

ASN.1 Information Object and Information Object Set definitions are used in table constraint specifications within ASN.1 types for automatic encoding/decoding of open type fields. A mapping is done of an Information Object Set declaration to XSD application information (appinfo) within generated types that reference the constraints.

The basic mapping is as follows:

ASN.1 Information Object Set definition:

```
InfoObjectSetName ClassName ::= {
   InfoObject declarations…
}
```

Generated XSD code:

```
<xsd:annotation>
   <xsd:appinfo>
      <ClassName>
         <object InfoObject attribute declarations/>
          ...
         </ClassName>
      </xsd:appinfo>
   </xsd:annotation>
```

The Information Object attribute declarations would be concrete instances of the attributes defined within the Class type. A separate row would be generated for each Information Object defined within the Information Object Set specification.

For example, the following is an Information Object Set specification using the NBAP-PROTOCOL-IES class defined above:

```
CommonSetupRequests NBAP-PROTOCOL-IES ::= {
 {ID           id-C-ID
  CRITICALITY  reject
  TYPE         C-ID
  PRESENCE     mandatory} |
 {ID           46
  CRITICALITY  reject
  TYPE         ConfigurationGenID
  PRESENCE     mandatory} |
 {ID           36
  CRITICALITY  ignore
  TYPE         SetupRqstFDD
  PRESENCE     mandatory},
    ...
 }
```

In this example, `id-C-ID` is an ASN.1 value defined as follows:

```
id-C-ID ProtocolIE-ID ::= 33
```

The generated XSD annotation for this declaration is as follows:

```
<xsd:annotation>
  <xsd:appinfo>
    <NBAP_PROTOCOL_IES>
      <object id="id_C_ID" type="C_ID"
       criticality="reject" presence="mandatory"/>
      <object id="43" type="ConfigurationGenID"
       criticality="reject" presence="mandatory"/>
      <object id="36" type="SetupRqstFDD"
       criticality="ignore" presence="mandatory"/>
     </NBAP_PROTOCOL_IES>
   </xsd:appinfo>
 </xsd:annotation>
```

Use of Mappings in Type Definitions

ASN.1 type definitions can be created that reference class fields and that are constrained by objects defined within an Information Object Set. The XSD mapping for these types contain normal element declarations for fixed type value fields and special “open type” elements for type fields.

The special open type elements will reference a generated complexType with a name in the following format:

<*Parent Type Assignment Name*>_<*Open Type Element Name*>_OpenType

The Information Object and Information Object Set definitions related to this type will be added as an annotation in the format described earlier. The generated type will be a choice between all of the different alternatives that make up the Information Object Set.

The basic mapping is as follows:

ASN.1 Definition:

```
TypeName ::= SEQUENCE {
   element1-name FixedTypeFieldRef ({TableConstraint}),
   element2-name FixedTypeFieldRef ({TableConstraint}{@key}),
   element3-name TypeFieldRef ({TableConstraint}{@key}),
      ...
}
```

Any combination of fixed type and type fields can be contained within the type definition.

Generated XSD code:

```
<xsd:complexType name="TypeName">
  <xsd:sequence>
    <xsd:element name="elem1Name" type="Field1Type"/>
    <xsd:element name="elem2Name" type="Field2Type"/>
    <xsd:element name="elem3Name"
        type="TypeName_elem3Name_OpenType"/>
      ...
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="TypeName_elem3Name_OpenType">
  <xsd:choice>
    <!-- this is the annotation for the info object set -->
    <xsd:annotation>
      <xsd:appinfo>
        <ClassName>
          <object attributes declarations/>
          ...
        </ClassName>
      </xsd:appinfo>
    </xsd:annotation>
    <!-- these are all of the various message alternatives
     (name and type) defined within the info object set →
    <xsd:element name="TypeName" type="TypeName"/>
      ...
    <!-- If info object set is extensible, message of unknown
     type is possible -->
    <xsd:any namespace="##other" processContents="lax"
     minOccurs="0"/>
  </xsd:choice>
</xsd:complexType>
```

In this case, the fixed type field types are obtained directly from the class definition. The type field is a reference to the generated open type field. The generated open type container type contains all of the information on the set of messages that is allowed to occupy the open type field.

An example showing all of this using the NBAP protocol is as follows:

The definition of an NBAP Protocol Information Element Field is as follows:

```
      ProtocolIE-Field ::= SEQUENCE {
        id
            NBAP-PROTOCOL-IES.&id({CommonSetupRequests}),
        criticality
            NBAP-PROTOCOL-IES.&criticality({CommonSetupRequests}{@id}),
        value
            NBAP-PROTOCOL-IES.&Value({CommonSetupRequests }{@id})
      }
```

This type allows any of the messages defined in the *CommonSetupRequests* information object set to be populated in the Value field. The id and criticality must match that of the defined message. The XSD definitions that are generated from this are as follows:

```
<xsd:complexType name="ProtocolIE_Field">
  <xsd:sequence>
    <xsd:element name="id" type=" ProtocolIE_ID"/>
    <xsd:element name="criticality" type="Criticality"/>
    <xsd:element name="value" type="ProtocolIE_Field_OpenType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ProtocolIE_Field_value_OpenType">
  <xsd:choice>
    <xsd:annotation>
      <xsd:appinfo>
        <NBAP_PROTOCOL_IES>
          <object id="id_C_ID" type="C_ID"
           criticality="reject" presence="mandatory"/>
          <object id="43" type="ConfigurationGenID"
           criticality="reject" presence="mandatory"/>
          <object id="36" type="SetupRqstFDD"
           criticality="ignore" presence="mandatory"/>
         </NBAP_PROTOCOL_IES>
        </xsd:appinfo>
      </xsd:annotation>
      <xsd:element name="C_ID" type="C_ID"/>
      <xsd:element name="ConfigurationGenID"
         type="ConfigurationGenID"/>
      <xsd:element name="SetupRqstFDD" type="SetupRqstFDD"/>
      <xsd:any namespace="##other" processContents="lax"
         minOccurs="0"/>
    </xsd:choice>
</xsd:complexType>
```

References

[1] *tML Guidelines for Mapping ASN.1 Syntax and Modules to XML Schemas*
Proposed Draft Standard – T1M1 Working Group – August 13-17, 2001