ASN1C

ASN.1 Compiler
Version 7.3
XML Schema Translator Users Guide
Reference Manual

Objective Systems, Inc. version 7.3 — January 2019

Copyright Notice

Author's Contact Information

Comments, suggestions, and inquiries regarding ASN1C may be submitted via electronic mail to info@obj-sys.com.

Table of Contents

Chapter 1. Overview of ASN.1-XSD Translation

The ASN1C code generation tool translates an Abstract Syntax Notation 1 (ASN.1) source file into computer language source files that allow ASN.1 data to be encoded/decoded. This release of ASN1C contains facilities to translate ASN.1 source specifications to an equivalent representation in World-Wide-Web Consortium (W3C) XML Schema Definition language (XSD).

A utility program is also provided to do the reverse translation from XSD to ASN.1 as specified in the ITU-T X.694 standard. This is the xsd2asn1 executable program located in the installation bin subdirectory. This program is described in the ***XML Schema to ASN.1 Conversion [http://www.obj-sys.com/docs/acv63/XSDHTML/ch04.html]*** section.

Chapter 2. Using the ASN2XSD command-line tool

The ASN.1-to-XSD translation capability is available in both the ASN1C compiler and in a separate tool named ASN2XSD. To generate XSD code using ASN1C, add -xsd to the ASN1C command-line or check the box in the ASN1C GUI wizard for generating an equivalent XSD file. The same result can be obtained by running ASN2XSD directly. In both cases, options are available for customizing the generated XSD code. These are summarized below for the ASN2XSD executable, but the same options can be used with ASN1C.

To test if ASN2XSD was successfully installed, enter asn2xsd with no parameters as follows (note: if you have not updated your PATH variable, you will need to enter the full pathname):

```
asn2xsd
```

You should observe the following display (or something similar):

```
ASN2XSD, Version 6.4.x
ASN.1 to XSD translation tool
Copyright (c) 2003-2011 Objective Systems, Inc. All Rights Reserved.

Usage: asn2xsd <filename> options

    <filename>              ASN.1 source file name

  options:
    -warnings               Output compiler warning messages
    -o <directory>          Output file directory
    -I <directory>          Import file directory
    -list                   Generate listing
    -appinfo [<items>]      Generate appInfo for ASN.1 items
                              <items> can be tags, enum, and/or ext
                              ex: -appinfo tags,enum,ext
                              default = all if <items> not given
    -attrs [<items>]        Generate non-native attributes for <items>
                              <items> is same as for -appinfo
    -targetns [<namespace>] Specify target namespace
                              <namespace> is namespace URI, if not given
                              no target namespace declaration is added
    -tables                 Generate XSD code for table constraints
    -stdout                 Output code to stdout
    -xer                    Generate XSD code corresponding to XER encoding
```

To use ASN2XSD, at a minimum, an ASN.1 source file must be provided. The source file specification can be a full pathname or only what is necessary to qualify the file. If directory information is not provided, the user's current default directory is assumed. If a file extension is not provided, the default extension ".asn" is appended to the name. Multiple source filenames may be specified on the command line to compile a set of files. The wildcard characters `*' and `%' are also allowed in source filenames (for example, the command `asn2xsd *.asn' will compile all ASN.1 files in the current working directory).

The source file(s) must contain ASN.1 productions that define ASN.1 types and/or value specifications. This file must strictly adhere to the syntax specified in ASN.1 standard ITU X.680.. The deprecated X.208 standard is not supported in ASN2XSD although the ANY type from that standard is recognized.

The following table lists all of the command line options related to ASN.1 to XSD translation.

Option	**Argument**	**Description**
-appInfo	tags enum ext	This option instructs asn2xsd to generate an <appinfo> section within the generated XSD file that contains additional ASN.1 specific information. This includes information about tags, enumerated types, or extension types. The argument is optional - if no argument is given information is generated for all of these items. It is also possible to specify multiple items by using a comma-separated list (for example, -appInfo tags,enum).
-attrs	tags enum ext	This option instructs asn2xsd to generate non-native attributes for elements within the generated XSD file that contain additional ASN.1 specific information. This includes information about tags, enumerated types, or extension types. The argument is optional - if no argument is given information is generated for all of these items. It is also possible to specify multiple items by using a comma-separated list (for example, -appInfo tags,enum).
-tables	<filename>	This option is used to generate additional code for the handling of table constraints as defined in the X.682 standard. See the *Generated Information Object Table Structures* section for additional details on the type of code generated to support table constraints.
-I	<directory>	This option is used to specify a directory that the compiler will search for ASN.1 source files for IMPORT items. Multiple -I qualifiers can be used to specify multiple directories to search.
-list	None	Generate listing. This will dump the source code to the standard output device as it is parsed. This can be useful for finding parse errors
-o	<directory>	This option is used to specify the name of a directoryto which all of the generated files will be written.
-pdu	<typeName>	Designate given type name to be a "Protocol Definition Unit" (PDU) type. This will cause a C++ control class to be generated for the given type. By default, PDU types are determined to be types that are not referenced by any other types within a module. This option allows that behavior to be overridden. The `*' wildcard character may be specified for <typeName> to indicate that all productions within an ASN.1 module should be treated as PDU types.
-targetns	<URI>	Specify URI for target namespace to be added to the generated XSD code. If this option is omitted, no target namespace declaration is added.
-warnings	None	Output information on compiler generated warnings.
-xer	None	Generate a schema corresponding to the XER encoding of the ASN.1

Chapter 3. ASN.1 to XML Schema Conversion

The ASN1C compiler contains the capability of generating corresponding XML Schema type definitions from ASN.1 types. This capability is also present in a free online tool (ASN2XSD) that may be accessed via the following URL:

http://www.obj-sys.com/asn2xsd.php [http://www.obj-sys.com/asn2xsd.php]

ASN.1 types are converted to XML Schema and ASN.1 values are converted to XML. ASN.1 value sets, which are essentially a set of constraints, get converted to facets in XML Schema.

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 each 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

Note

This section decribes the mapping of ASN.1 types to XSD types. The mapping presented here corresponds to the XML encoding that ASN1C produces when using the *-xml* option. This is the default mapping followed by ASN2XSD. This section is *not* relevant to XER encoding and therefore is irrelevant if you have specified the *-xer* command line option for ASN1C or ASN2XSD.

Each ASN.1 type is mapped to a corresponding XSD type. Some ASN.1 types have a natural mapping to an XSD type (for example, an ASN.1 BOOLEAN type maps to an xsd:boolean type). In other cases, custom types were needed because a natural mapping did not exist within XSD (for example, there was no direct mapping for an ASN.1 BIT-STRING type). These custom types can be found in the low-level ASN.1 XML schema definitions library at the following URL:

http://www.obj-sys.com/v1.0/XMLSchema/asn1.xsd

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 ASN2XSD 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>
```

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>
```

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="xsd: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="xsd: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 *-attrs enum* command-line option is specified and the enumerated items contain numbers (i.e do not follow the standards sequence), then an *asn1:value* attribute is added to the type to allow an application to map the enumerated identifiers to numbers. If an *asn1:value* attribute 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), etc.)
```

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 asn1:value attributes added to the enumeration items above are an example of *non-native attributes*. These are attributes that are not defined within the XML Schema standard but that may be added to provide additional information about an element contained within the schema. Conformant XML schema processors should ignore these attributes. They are only added to the generated code if the *-attrs enum* option is added to the ASN1C command-line (or *-attrs* option with no qualifiers)

It is also possible to generate an "application information" (appinfo) section within the generated schema containing information on the enumerated values. This is done using the *-appinfo enum* option (or *-appinfo* with no qualifiers). The generated code with <appinfo> would be as follows:

```
<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>
```

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

```
<xsd: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 pattern is 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-name"/>
      ...
   </xsd:sequence>
</xsd: complexType>
```

SET

An ASN.1 SET is a constructed type consisting of a series of element definitions that must appear in any order. This is very 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>
```

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

In this definition, the element name is based on the name of the element type. The element type 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 maxOccurs="unbounded">
      <xsd:element name="elementName" type="ElementType"/>
   </xsd:sequence>
</xsd:complexType>
```

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>typedef struct {
```

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.

If the CHOICE type is extensible (i.e., contains an ellipses marker ...), a special element will be inserted to allow an unknown alternative to be validated. This 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 choice alternative element is a defined element or an extension element.

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
```

A reference to an *Open Type* within a SEQUENCE or CHOICE construct is converted into an XSD *any* element type. Note that the conversion is only done if the element is in one of these constructs. An open type declaration on its own has no equivalent XSD type and is therefore ignore

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

In this case, any valid XML instance can be used in the type. Note that the ASN.1 element name (*anOpenType*) is ignored.

If the open type is bound a by a relational table constraint and -tables was specified on the command-line, an XSD choice is created that contains all of the possible types that may appear in the field. This is further described in the section on mapping items from table constraints.

Tagged Type

In ASN.1, it is possible to create new custom types 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 either non-native attributes or an application information annotation (*appinfo*) to the generated XSD type describing the ASN.1 tags.

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 asn1:TagInfo type in the Objective Systems XSD class library.

For the non-native attributes case (specified by adding -attrs tags or -attrs with no qualifiers to the ASN1C command-line), 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>
```

For the appInfo case (specified by adding -appinfo tags or -appinfo with no qualifiers to the ASN1C command-line), the mapping is as follows:

```
<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 Type

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>
```

Elements with Table Constraints

The ITU-T ASN.1 X.681 and X.682 standards specify a table-driven 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. Elements within types that are constrained in this way result in a special mapping to be done. This is only done when the *-tables* option is specified.

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 an anonymous <choice> complexType that will contain an alternative for each of the type fields listed in the referenced information object set that are allowed for the type.

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">
            <xsd:complexType>
               <xsd:choice>
                  <xsd:element name="infoObjectName" type="InfoObjectType">
                  ...
               </xsd:choice>
            </xsd:complexType>
         </xsd:element>
      </xsd:sequence>
   </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 <choice> container type contains all of the possible elements allowed by the table constraint for the open type field.

Example

This example is from a modified version of the 3GPP NBAP ASN.1 specification. Thsi protocol makes heavy use of classes, information objects, and information object sets.

One example of this is the NBAP *InitiatingMessage* type which is defined is as follows:

```
   InitiatingMessage ::= SEQUENCE {
      procedureID NBAP-ELEMENTARY-PROCEDURE.
         &procedureID ({NBAP-ELEMENTARY-PROCEDURES}),

      criticality NBAP-ELEMENTARY-PROCEDURE.
         &criticality ({NBAP-ELEMENTARY-PROCEDURES}{@procedureID}),

      messageDiscriminator NBAP-ELEMENTARY-PROCEDURE.
         &messageDiscriminator ({NBAP-ELEMENTARY-PROCEDURES}{@procedureID}),

      transactionID TransactionID,

      value NBAP-ELEMENTARY-PROCEDURE.
         &InitiatingMessage ({NBAP-ELEMENTARY-PROCEDURES}{@procedureID})
   }
```

A simplified version of the NBAP-ELEMENTARY-PROCEDURES information object set that defines the contents of some of the element fields is as follows:

```
   NBAP-ELEMENTARY-PROCEDURES NBAP-ELEMENTARY-PROCEDURE ::= {
      radioLinkSetupFDD |
      radioLinkSetupTDD,
      ...
   }
```

The NBAP-ELEMENTARY-PROCEDURE class is defined as follows:

```
   NBAP-ELEMENTARY-PROCEDURE ::= CLASS {
      &InitiatingMessage,
      &SuccessfulOutcome        OPTIONAL,
```

```
   &UnsuccessfulOutcome      OPTIONAL,
   &Outcome                  OPTIONAL,
   &messageDiscriminator MessageDiscriminator,
   &procedureID ProcedureID  UNIQUE,
   &criticality Criticality  DEFAULT ignore
}
WITH SYNTAX {
   INITIATING MESSAGE        &InitiatingMessage
   [SUCCESSFUL OUTCOME       &SuccessfulOutcome]
   [UNSUCCESSFUL OUTCOME     &UnsuccessfulOutcome]
   [OUTCOME                  &Outcome]
   MESSAGE DISCRIMINATOR     &messageDiscriminator
   PROCEDURE ID              &procedureID
   [CRITICALITY              &criticality]
}
```

Finally, the information objects that are referenced in the information object set are as follows:

```
-- *** RadioLinkSetup (FDD) ***
radioLinkSetupFDD NBAP-ELEMENTARY-PROCEDURE ::= {
   INITIATING MESSAGE    RadioLinkSetupRequestFDD
   SUCCESSFUL OUTCOME    RadioLinkSetupResponseFDD
   UNSUCCESSFUL OUTCOME  RadioLinkSetupFailureFDD
   MESSAGE DISCRIMINATOR common
   PROCEDURE ID { procedureCode id-radioLinkSetup, ddMode fdd }
   CRITICALITY  reject
}

-- *** RadioLinkSetup (TDD) ***
radioLinkSetupTDD NBAP-ELEMENTARY-PROCEDURE ::= {
   INITIATING MESSAGE     RadioLinkSetupRequestTDD
   SUCCESSFUL OUTCOME     RadioLinkSetupResponseTDD
   UNSUCCESSFUL OUTCOME   RadioLinkSetupFailureTDD
   MESSAGE DISCRIMINATOR  common
   PROCEDURE ID  { procedureCode id-radioLinkSetup, ddMode tdd }
   CRITICALITY   reject
}
```

After working through the various layers, ASN2XSD is able to produce an XSD definition that provides fixed type references for the procedureID, criticality, and messageDiscriminator elements. An anonymous choice is produced under the value element that defines all of the types that may be used in the open type eleemnt. This is the resulting XSD definition:

```
<xsd:complexType name="InitiatingMessage">
   <xsd:sequence>
      <xsd:element name="procedureID" type="NBAP-CommonDataTypes:ProcedureID"/>
      <xsd:element name="criticality" type="NBAP-CommonDataTypes:Criticality"/>
      <xsd:element name="messageDiscriminator" type="NBAP-CommonDataTypes:MessageDiscri
      <xsd:element name="transactionID" type="NBAP-CommonDataTypes:TransactionID"/>
      <xsd:element name="value">
         <xsd:complexType>
            <xsd:choice>
               <xsd:element name="radioLinkSetupFDD" type="NBAP-PDU-Contents:RadioLinkS
               <xsd:element name="radioLinkSetupTDD" type="NBAP-PDU-Contents:RadioLinkS
            </xsd:choice>
```

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

Chapter 4. XML Schema to ASN.1 Conversion

The version 5.8 release of ASN1C contains a separate command-line utility program that translates an XML Schema Definitions (XSD) source file into an equivalent ASN.1 source file. This conversion process is based on the ITU-T X.694 standard which specifies a mapping from XSD to ASN.1.

The mapping specified in the standard consists of two parts:

1. A mapping of XSD elements, types, and attributes to equivalent ASN.1 items, and

2. The addition of "extended-XER" (E-XER) annotations to allow the ASN.1 source file to act as a stand-alone schema for use in both XML and ASN.1 applications.

The ASN1C translation process supports only the first item at this time. The main goal of the translation process is to get the XML schema into a form that allows the generation of efficient binary encoders/decoders that utilize the ASN.1 encoding rules. Option 1 supports this goal by creating an ASN.1 file that can be used with any ASN.1 compiler product or tool that supports standard ASN.1 syntax.

It should also be noted that the translation process of going from ASN.1 to XSD described in the previous section is not "round-trippable" with the XSD to ASN.1 process described in this section. That is to say, one cannot start with an XSD file (for example) and translate it to ASN.1 using this tool and then translate that file back to XSD and expect the final XSD file to be the same as the original. Certain naming conventions that are utilized make this type of round-trip conversion process very problematic. It is therefore a one-way process - a user must work either in XSD or ASN.1 and then use the tools to get an equivalent representation in the alternative form.

Running the XSD-to-ASN.1 Translation Tool from the Command-line

The XSD-to-ASN.1 translation tool is the *xsd2asn1* utility program that can be found in the bin subdirectory of an ASN1C installation. To test if the compiler was successfully installed, enter xsd2asn1 with no parameters as follows (note: if you have not updated your PATH variable, you will need to enter the full pathname):

```
xsd2asn1
```

You should observe the following display (or something similar):

```
XSD2ASN1, Version 0.2.x
Copyright (c) 2005 Objective Systems, Inc. All Rights Reserved.
```

Usage:

```
xsd2asn1 <filename> options
```

<filename> XML schema or WSDL source file name(s). Multiple filenames may be specified. * and ? wildcards are allowed.

options:

```
-config <file> Specify schema bindings file
-o <directory> Output file directory
-I <directory> Import file directory
```

```
-all Compile all dependent files
-warnings Output compiler warning messages
```

The XSD source file specification can be a full pathname or only what is necessary to qualify the file. If directory information is not provided, the user's current default directory is assumed. Multiple source filenames may be specified on the command line to compile a set of files. The wildcard characters `*' and `?' are also allowed in source filenames (for example, the command `xsd2asn1 *.xsd' will translate all XSD files in the current working directory to ASN.1).

The following table lists all of the command line options supported in this version of the tool.

Option	**Argument**	**Description**
-all	None	This option is used to specify that all dependent files should be translated to ASN.1 as well as the main file being compiled. This includes all files included using XSD <include> and <import> directives.
-config	<filename>	This option is used to specify the name of a file containing configuration information for the source file(s) being parsed. A full discussion of the contents of a configuration file is provided in the XML Schema Binding File section.
-I	<directory>	This option is used to specify a directory that the compiler will search for ASN.1 source files for IMPORT items. Multiple -I qualifiers can be used to specify multiple directories to search.
-o	<directory>	This option is used to specify the name of a directory to which all of the generated files will be written.
-warnings	None	Output information on compiler generated warnings.

XML Schema Binding File

The schema bindings file is an XML file that allows customizations of certain aspects of the XSD-to-ASN.1 translation process. It is different from command-line switches in that it provides a way to associate configuration items with specific XSD information items within a schema or set of schemas.

For this release of XSD2ASN1, the only useful configuration item that can be specified is the location of individual source files for schemas included using the <xsd:include> and <xsd:import> declarations. These declarations allow a schemaLocation attribute to be specified, but this attribute does not necessarily have to contain a full path to the referenced file (it is described in the standard as only providing a hint for a schema processor to help in locating the file). Since third-party schemas cannot always be editied by developers, the schema binding file provides a mechanism to bind the file location information to the schema without requiring edits to the original schema.

At the outer level of the schema binding file is a <bindings> element. This is a container element that holds all of the binding elements for specific schemas. There can be one and only one <bindings> element in a schema bindings file. This element contains a mandatory "version" attribute that specifies the verion of the schema binding language

in use. For this version of XSD2ASN1, the only supported version is 1.0. Therefore, the outer level of the schema bindings file will always look like this:

```
<bindings version="1.0">
.. specific binding elements here ..
</bindings>
```

The only element supported below the <bindings> level is the <schemaBindings> element. This allows the association of configuration items with a specific schema. The schema is specified using the namespace attribute which identifies the schema using its target namespace.

The actual location of an include or import file can then be specified using the binding file <sourceFile> element. The content of this element is the full or relative pathname to the schema file on a local computer (access to a web URI is not supported at this time).

As an example, suppose a schema contained the following import directive:

```
<xsd:import namespace="http://example.com//ImportElement"
schemaLocation="aImportElement.xsd"/>
```

It is possible to specify a different path for the *aImportElement.xsd* file if it did not reside in the current working directory using a schema binding file. Assume it was located in the c:\importSchemas directory. The binding file would be as follows:

```
<bindings version="1.0">
   <schemaBindings namespace="http://example.com//ImportElement">
      <sourceFile>c:\importSchemas\aImportElement.xsd</sourceFile>
   </schemaBindings>
</bindings>
```

XSD-to-ASN.1 Information Item Mappings

All XSD to ASN.1 information item mappings are as specified in the ITU-T X.694 standard, a free copy of which is available at the following URL:

http://www.itu.int/ITU-T/studygroups/com17/languages/X694pdf

The only non-standardized item is the module name used for the generated ASN.1 module. XSD2ASN1 assigns module name as follows:

1. It will attempt to use the last delimitted item at the far right in the target namespace declaration. For example, if the target namespace URI is "http://foo/bar", "Bar" will be used as the module name (note the first letter was capitialized as per X.694 naming rules)

2. If the target namespace is not in a standard URI format or if the last delimitted name contains special chanracters or is very long, then the name of the original XSD source file is used.