ASN1C

ASN.1 Compiler
Version 6.0
XML Schema Translator
User’s Manual

Objective Systems, Inc. - October 2006

Author’s Contact Information:

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

CHANGE HISTORY

Date	Author	Version	Description
10/15/2006	ED	6.0	Initial version

TABLE OF CONTENTS

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* section.

Using the Compiler

Running ASN1C from the Command-line

The ASN1C compiler distribution contains a command-line compiler executable as well as a graphical user interface (GUI) wizard (Windows version only) that can aid in the specification of compiler options. This section describes how to run the command-line version. Information on the GUI wizard can be found in the *ASN1C C/C++ Users' Guide*.

To test if the compiler was successfully installed, enter `asn1c` with no parameters as follows (note: if you have not updated your PATH variable, you will need to enter the full pathname):

```
      asn1c
```

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

```
ASN1C Compiler, Version 5.8x
Copyright (c) 1997-2005 Objective Systems, Inc. All Rights Reserved.

Usage: asn1c <filename> <options>

    <filename>          ASN.1 source file name(s).  Multiple filenames
                          may be specified.  * and ? wildcards are allowed.

  language options:
    -c                  generate C code
    -c++                generate C++ code
    -c#                 generate C# code
    -java               generate Java code
    -xsd [<filename>]   generate XML schema definitions

  encoding rule options:
    -ber                generate BER encode/decode functions
    -cer                generate CER encode/decode functions
    -der                generate DER encode/decode functions
    -per                generate PER encode/decode functions
    -xer                generate XER encode/decode functions
    -xml                generate XML encode/decode functions

  basic options:
    -asnstd <std>       set standard to be used for parsing ASN.1
                        source file. Possible values - x208, x680, mixed
                        (default is x680)
    -compact            generate compact code
    -compat <version>   generate code compatible with previous
                        compiler version. <version> format is
                        x.x (for example, 5.3)
    -config <file>      specify configuration file
    -depends            compile main file and dependent IMPORT items
    -I <directory>      set import file directory
    -lax                do not generate constraint checks in code
    -list               generate listing
    -nodecode           do not generate decode functions
    -noencode           do not generate encode functions
    -noIndefLen         do not generate indefinite length tests
    -noOpenExt          do not generate open extension elements
    -notypes            do not generate type definitions
    -o <directory>      set output file directory
    -pdu <type>         designate <type> to be a Protocol Data Unit (PDU)
                        (<type> may be * to select all type definitions)
    -print [<filename>] generate print functions
    -shortnames         reduce the length of compiler generated names
    -trace              add trace diag msgs to generated code
    -uniquenames        resolve name clashes by generating unique names
    -warnings           output compiler warning messages

  C/C++ options:
    -hfile <filename>   C or C++ header (.h) filename
                          (default is <ASN.1 Module Name>.h)
    -cfile <filename>   C or C++ source (.c or .cpp) filename
                          (default is <ASN.1 Module Name>.c)
    -genBitMacros       generate named bit set, clear, test macros
    -genInit            generate initialization functions for all types
    -genFree            generate memory free functions for all types
    -genMake            generate makefile to build generated code
    -maxlines [<num>]   set limit of number of lines per source file
                        (default value is 50000)
    -oh <directory>     set output directory for header files
    -static             generate static elements (not pointers)
    -w32                generate code for Windows O/S (default=GNU)

  Java options:
    -pkgpfx <text>     Java package prefix
    -pkgname <text>    Java package name
    -dirs              output Java code to module name dirs
    -genjsources       generate <modulename>.mk for list of java files
    -getset            generate get/set methods and protected member vars
    -genbuild          generate build script
    -compare           generate comparison functions

  C# options:
    -nspfx <text>      C# namespace prefix
    -namespace <text>  C# namespace name
    -dirs              output C# code to module name dirs
    -gencssources      generate <modulename>.mk for list of C# files

  pro options:
    -events            generate code to invoke SAX-like event handlers
    -stream            generate stream-based encode/decode functions
    -tables            generate table constraint functions
    -strict            do strict checking of table constraint conformance
    -prtToStr [<filename>]
```

```
                       generate print-to-string functions (C/C++)
    -prtToStrm [<filename>]
                       generate print-to-stream functions (C/C++)
    -genTest  [<filename>]
                       generate sample test functions
    -reader            generate sample reader program
    -writer            generate sample writer program
    -compare [<filename>]
                       generate comparison functions (C/C++)
    -copy [<filename>] generate copy functions (C/C++)
    -maxcfiles         generate separate file for each function (C/C++)

  XSD options:
    -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
```

To use the compiler, 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 'asn1c *.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 '-asnstd x208' command-line option should be used to parse files based on the 1990 ASN.1 standard (x.208) or that contain references to ROSE macro specifications.

The following table lists all of the command line options related to ASN.1 to XSD translation Options for code generation for other computer languages are doucmented in the respective language user guide documents.

Option	**Argument**	**Description**
-appInfo	tags enum ext	This option instructs the compiler 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).

-asnstd	x208 x680 mixed	This option instructs the compiler to parse ASN.1 syntax conforming to the specified standard. 'x680' (the default) refers to modern ASN.1 as specified in the ITU-T X.680-X.690 series of standards. 'x208' refers to the now deprecated X.208 and X.209 standards. This syntax allowed the ANY construct as well as unnamed fields in SEQUENCE, SET, and CHOICE constructs. This option also allows for parsing and generation of code for ROSE OPERATION and ERROR macros and SNMP OBJECT-TYPE macros. The 'mixed' option is used to specify a source file that contains modules with both X.208 and X.680 based syntax.
-attrs	tags enum ext	This option instructs the compiler 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).
-compat	<versionNumber>	Generate code compatible with an older version of the compiler. The compiler will attempt to generate code more closely aligned with the given previous release of the compiler. <versionNumber> is specified as x.x (for example, -compat 5.2)
-config	<filename>	This option is used to specify the name of a file containing configuration information for the source file being parsed. A full discussion of the contents of a configuration file is provided in the *Compiler Configuration File* section.
-depends	None	This option instructs the compiler to generate a full set of header and source files that contain only the productions in the main file being compiled and items those productions depend on from IMPORT files.
-genTables -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.
-xsd	<filename>	This option instructs the compiler to generate an equivalent XML Schema Definition (XSD) for each of the ASN.1 productions in the ASN.1 source file. The definitions are written to the given filename or to <modulename>.xsd if the filename argument is not provided.

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/asn2xsdform.shtml

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 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. 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>
```

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 annotation 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 `asn1:bitno` 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 named bit numbers. 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: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 generated if the bits are sequentially numbered starting at zero.

Example

The following ASN.1 BIT STRING type:

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

maps to the following XSD type when *-attrs* is specified on the command-line:

```
<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>
```

If *-attrs* is omitted, the generated type will not contain the *asn1:bitno* attributes.

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="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>
```

Example

The following ASN.1 enumerated type:

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

maps to the following XSD type when *-attrs* is specified on the command-line:

```
<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 *-attrs* command-line option was not specified or if the identifiers in the enumerated type did not contain numbers (i.e. if the type was 'ENUMERATED (blue, green, red)'), then the `asn1:value` attributes would not be added to the code above.

Extensible ENUMERATED

An ENUMERATED type may include an extensibility marker (…) to indicate that enumerated items not defined in the current set may be added at a later time. The generated XSD type in this case must allow items outside of the base set to not cause a validation error. This is accomplished by adding a union with an *xsd:token* type to the generated type definition:

ASN.1 production:

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

Generated XSD code:

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

```
<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>
```

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 {
   anIntINTEGER 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 {
   anIntINTEGER 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.

If the *-attrs ext* command-line option is specified (or *-attrs* with no qualifiers), 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>
```

In this case, the extElm extension element contains the `asn1:description="extension element"` attribute which allows an application to detect that it is an extension element. Also note that all extension elements are marked optional (*minOccurs="0"*) whether they are declared to be optional or not. That is because an instance of a message conforming to an earlier version of the message protocol may not contain these elements.

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

In this definition, the element name is the hard-coded value ‘element’. 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>
```

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

Note that the element name is the hard-coded name ‘element’. 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 maxOccurs=”unbounded”>
      <xsd:element name="myString" type="xsd:string"/>
   </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="element" 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>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 ignored.

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.

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>
```

Mapping of ASN.1 Information Objects

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. 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>
```

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.

Index