ASN2TXT v2.9

User's Manual

Objective Systems, Inc. — February 2019

The software described in this document is furnished under a license agreement and may be used only in accordance with the terms of this agreement.

Copyright Notice

Copyright ©1997–2019 Objective Systems, Inc. All rights reserved.

This document may be distributed in any form, electronic or otherwise, provided that it is distributed in its entirety and that the copyright and this notice are included.

Author's Contact Information

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

Table of Contents

Revision History....8
Overview of ASN2TXT....10
Using ASN2TXT....11
  Installation....11
  Installing on a Windows System....11
  Installing on a UNIX System....11
  Command-line Options....12
    Input and Output Encoding Options....14
    Common Options....15
    XML/JSON Options....17
    CSV Options....18
  Using the GUI....20
  Configuration Files....28
  ASN.1 Tag Path Filtering....30
Using the ASN2TXT DLLs....33
  Setup for use from C++....33
  Cleanup in C++....33
  Performing Translations....34
  Setup for use from C....36
  Cleanup in C....37
  Setup for use from .Net....37
  Cleanup in .Net....38
  Licensing....38
  Document Object Model (DOM)....39
Organizing CSV Output....42
  Exploding and Trimming CSV Output....42
  Filtering CSV Data....44
    Filter Alphabet....45
    Filter Configuration Syntax....45
    Filter Paths....46
    Wildcard Selections....48
    Further Examples....49
    Filtering Best Practices....49
ASN.1 to XML Type Mappings....51
  General Mapping without ASN.1 Schema Information....51
  General Mapping with ASN.1 Schema Information....52
  Specific ASN.1 Type to Value Mappings....53
ASN.1 to JSON Type Mappings....62
ASN.1 to CSV Type Mappings....62
  Mapping Top-Level Types....63
  Mapping Simple Types....64
  Mapping Complex Types....65
    CHOICE....66

Simple SEQUENCEs and SETs....66
Mapping Nested Types....67
SEQUENCE in a SEQUENCE....67
CHOICE in a SEQUENCE....67
SEQUENCE OF in a SEQUENCE....68
Data Conversion....68
SEQUENCE OF in a SEQUENCE....68
Other Nested Data Types....69
OPTIONAL and DEFAULT Elements....70

Revision History

- July 2015 — Initial release of documentation (2.5.0)
- February 2016 – Add JSON support, 3GPP TS 32.297 support, and other minor updates.
- October 2016 – Add DOM support.
- March 2017 – Add DOM support to convert from XML to binary (BER/PER) form.
- May 2017 – Add information on new translations :

  XML/XER → DER, Binary/XML/XER → CXER, BER → DER
- February 2018 – Version and copyright updates, as well as updated GUI screen shots.
- January 2019 – Version and copyright updates. Add reference to X.697 JSON Encoding Rules (JER).

Overview of ASN2TXT

ASN2TXT is a command-line tool and embeddable library that translates ASN.1 data encoded in the Basic, Canonical, Distinguished, or Packed encoding rules into various text formats suitable for ingestion into spreadsheets, databases, or other text processing tools. As of version 2.5, ASN2TXT supports converting ASN.1 data to XML, JSON, and comma-separated value (CSV) data formats. Version 2.7 adds support for reverse translations from XML to binary formats (BER, DER, PER and U-PER). Support has also been added for translation to Canonical XER (CXER).

Conversions to XML support both an Objective Systems custom format as well as the XML Encoding Rules standard as described in ITU-T standard X.693. Conversions from all ASN.1 binary encodings (BER, CER, DER, PER) are supported by ASN2TXT.

Conversions to JSON conform to ITU-T X.697 JSON Encoding Rules (JER). (Note: earlier versions followed a proprietary format, which is documented, along with the differences from X.697, at

https://www.obj-sys.com/docs/JSONEncodingRules.pdf.)

Conversions to CSV are done by a custom transformation, since no standard for converting ASN.1-encoded data to CSV exists. Conversions from BER, CER, and DER are supported at the time of this release; PER is not supported.

Users can run ASN2TXT as a command-line application or build their own applications using the shared library provided in the package. A GUI Wizard application is also supplied. The library API is documented separately (such as for its use with DOM operations), but examples of its use are provided in this document as well.

The ASN2TXT binary-to-XML translation capability is also available online at the following URL:

https://www.obj-sys.com/webtools/asn2xml.php

(user registration is required).

Using ASN2TXT

Installation

ASN2TXT comes packaged as an executable installation program for Windows or a `.tar.gz` archive for UNIX systems. The package is comprised of the following directory tree:

```
asn2txt_v290
|
+-asn1specs
|
+-bin
|
+-doc
|
+-sample
|
+-sample_dll
|
+-sample_per
```

The `bin` subdirectory contains the `asn2txt` executable. The `asn1specs` directory contains specifications used by some of the sample programs in the `sample` directory. This document is found in the `doc` directory.

Installing on a Windows System

To install ASN2TXT on a Windows system, simply double-click the executable installer program. Selecting the default installation options will install ASN2TXT in `c:\asn2txt_v29x` where x would be replaced with a minor version number.

Installing on a UNIX System

To install ASN2TXT on a UNIX system, simply unzip and untar the `.tar.gz` archive. The program may be unpacked in any directory in which the user has permissions. No installation program is

available to install ASN2TXT to `/usr/local` or other common installation paths, but it is not difficult to manually add links if needed.

Command-line Options

Invoking `asn2txt` without any options will show a usage message that contains the command-line options:

```
ASN2TXT, version 2.9.0
ASN.1 to text formatter
Copyright (c) 2004-2019 Objective Systems, Inc.  All Rights Reserved.

Usage: asn2txt <input files> options

  <input files>            ASN.1 message file name (wildcards are okay)

  Input encoding options:
    -in <encode_rule>
      ber                  Decode from BER (basic encoding rules)
      per                  Decode from PER (aligned packed encoding rules)
      uper                 Decode from U-PER (unaligned packed encoding rules)
      xml                  Decode from XML
      xer                  Decode from XER (XML encoding rules)
      cxer                 Decode from CXER (canonical XML encoding rules)

  Output encoding options:
    -out <encode_rule>
      xml                  Encode to XML type
      xer                  Encode to XML XER type
      cxer                 Encode to CXER type
      json                 Encode to JSON type
      csv                  Encode to CSV type
      ber                  Encode to BER type
      per                  Encode to PER type
      uper                 Encode to U-PER type
      der                  Encode to DER type
```

```
  Common options:
    -schema <filename>     ASN.1 definition file name(s)
    -config <filename>     Configuration file name
    -I <directory>         Import ASN.1 files from <directory>
    -pdu <typename>        Message PDU type name
    -o <filename>          Output filename (use "<base>.xml" for XML batch output)
    -bcdhandling <default|none|bcd|tbcd>
                           Define handling of OCTET STRINGs declared to be
                             binary coded decimal (BCD) strings
    -oidformat <oidDefValue|namedNumbers|numbersOnly|dottedNamedNumbers|
dottedNumbers>
                           Define format of Object Identifier display
    -noopentype            Disable automatic open type decoding
    -paddingbyte <hexbyte> Additional padding byte
    -bitsfmt <hex|bin>     BIT STRING content output format
    -inputFileType <binary|hextext|base64|pem|pcap|pcapng>
                           Format of data in input file
    -skip <num>            Skip <num> bytes between messages
    -skipend <num>         Skip <num> bytes at end of message
    -headerOffset <num>    Skip the first <num> bytes in a data file
    -tagfilter <filename>  Tag filter file name
    -q                     Turn off all output except errors
  XML options:
    -ascii                 Print out ASCII for printable hex values
    -emptyoptionals        Insert empty XML elements in place of
                             missing optional elements
    -emptydefault          Insert XML elements with default values in place
                             of missing elements with default values
    -nowhitespace          Remove all whitespace between elements
    -noxmlcomments         Do not include header comment or message count in output
    -rootElement <element> Root Element Name
    -skip-bad-records      Skip messages which could not be decoded
    -headers <ts-32-297|length4|eth|tcpip|tpkt>
                           Headers to be processed before start of encoded
                           data.  Multiple headers can be specified using +
                           character (for example, eth+tcpip).  Header
```

```
                          types are:
                          ts-32-397 - 3GPP TS 32.297 CDR records
                          length4 - 4-byte length header
                          eth – Ethernet
                          tcpip – TCP/IP
                          tpkt - TPKT
  CSV options:
    -addInputFn           Add the input filename to the output CSV file
    -minLevel <num>       Set the minimum output depth
    -maxLevel <num>       Set the maximum output depth
    -noquotes             Do not quote strings in output file
    -outdir <directory>   Specify the output directory
    -padFields            Pad fields with data that would otherwise be empty
    -prefix               Prefix output filenames with input filenames
    -s <separator>        Field separator
    -seqsep <separator>   SEQUENCE OF field separator (defaults to '|')
    -separate-lines       Output fields of a SEQUENCE OF on separate lines
```

The following sections summarize the command-line options.

Input and Output Encoding Options

The following input and output encoding options describe the various transformations.

Option	Arguments	Description
-in	ber	Decode input from a BER file.
	per	Decode input from a PER file.
	uper	Decode input from a U-PER file.
	xml	Decode input from an XML file.
	xer	Decode input from an XER file.
	cxer	Decode input from a CXER file.
-out	xml	Encode/write output to an XML file.
	xer	Encode/write output to an XER file.
	cxer	Encode/write output to a CXER file.

Option	Arguments	Description
	json	Encode/write output to a JSON file.
	csv	Encode/write output to a CSV file.
	ber	Encode/write output to a BER file.
	per	Encode/write output to a PER file.
	uper	Encode/write output to a U-PER file.
	der	Encode/write output to a DER file.

Common Options

The following options are common to all transformations.

Option	Arguments	Description
<filename>		<filename> is the name of the input message to decode. This element is *required*. The use of wildcards (*e.g.** and ?) is supported.
-schema	<filename>	This option is *required* when using CSV or decoding PER data. When converting BER data to XML or JSON, a schema is optional. Without a schema, ASN2TXT will convert the data to a non-standard encoding, using tag names.
-config	<filename>	Allows an Obj-Sys configuration file to be used in translation. Configuration settings can be used to apply options to specific items within a schema.
-bcdhandling	<default | none | bcd | tbcd>	Define handling of OCTET STRINGs declared to be binary coded decimal (BCD) strings. By default, types declared as BCD or TBCD strings will be translated as such. <none> forces translation to be performed as usual, while <bcd> and <tbcd> force their respective formatting on such OCTET STRINGs.
-bitsfmt	<hex | bin>	Specify how BIT STRING items are formatted. By default they are expressed as hexadecimal strings; use bin to express them as binary strings instead.
-inputFileType	<binary | hextext | base64 |	Specify how the input data is formatted. By de-

Option	Arguments	Description
	pem | pcap | pcapng>	fault ASN2TXT will assume that the input data is binary, but it can also decode hexadecimal, base64, or PEM encoded data. Whitespace in the input is ignored when hextext is specified.
-noopentype		Disables the conversion of open types in the output. This is the default behavior when converting BER to CSV.
-oidformat	<oidDefValue | namedNumbers | numbersOnly | dottedNamedNumbers | dottedNumbers>	Define format of Object Identifier display. By default (or using <oidDefValue>), the value is displayed as it was defined. Using <namedNumbers> or <numbersOnly>, the value is displayed as such (for example “iso(1)” or “1”, respectively) and includes as many arcs as possible.
-I	<directory>	Specify directory to import additional ASN.1 files from.
-paddingbyte	<hexbyte>	<hexbyte> is the hexadecimal value of a padding byte that may appear in the input message. Call data records (CDRs) are commonly continuously dumped to files by telephony equipment. If no information is available, the records are often padded by 0x00 or 0xFF bytes. The default padding byte is 0x00. <hexbyte> may be formatted with or without a 0x prefix.
-pdu	<typename>	<typename> is the name of the PDU data type to be decoded. This option is necessary when the top-level data type is ambiguous. It is also required when converting PER data.
-o	<filename>	Specify the output XML or JSON <filename> instead of writing output to standard out. Set <filename> to “<base>.xml” to specify batch output when converting multiple files for XML.
-skip	<num>	Specify the number of bytes to skip at the start of each message in the data file.
-skipend	<num>	Specify the number of bytes to skip at the end of each message in the data file.
-headerOffset	<num>	Specify the number of bytes to skip at the start of the data file.

-I	<directory>	Specify directory to import additional ASN.1 files from.
-tagFilter	<filename>	Specify the file that contains the tag filtering path, a concatenated list of ASN.1 tag values that target specific elements within a message. See the “ASN.1 Tag Path Filtering” section.
-q		Operate in a “quiet” mode more suitable for batch processing. Informational messages are limited and only error output will be reported.

XML/JSON Options

The following options can be used when converting to XML or JSON.

Option	Arguments	Description
-ascii		If all bytes in a decoded value are within the ASCII character range, display as standard text. Otherwise display as formatted hexadecimal text. This option only has meaning if BER data is decoded without a schema file.
-emptyDefault		Insert an element with a default value as specified in the schema at the location of a missing element in the instance.
-emptyOptionals		Insert an empty element at the location of a missing element in the schema that was declared to be optional.
-nowhitespace		Do not generate any whitespace (blanks and newline characters) between elements. This makes the generated document more compact at the expense of readability.

-rootElement	<name>	Specify the root element <name> used to wrap the entire XML message at the outer level. This makes it possible to create an XML document for an ASN.1 file containing multiple individually encoded binary messages (a common feature of many Call Detail Record ASN.1 formats). The <name> argument is optional. If not provided, no root element is added to the XML output. If the -rootElement option is not specified, the message is wrapped in an element named “message” by default.
-noxmlcomments		Remove XML header comment and message/block count from output. (XML only)
-skip-bad-records		This option enables more thorough detection of badly formed records and attempts to skip such records. This can occur if an unrecognized tag is encountered, for example. In some cases, it is impossible to continue translation after a bad record, such as when an incorrect length value was encoded.
-headers	<ts-32-297 | length4 | eth | tcpip | tpkt>	Specify a message file containing various types of headers, such as 3GPP TS 32.297 CDR records or Ethernet or TCP-IP headers.

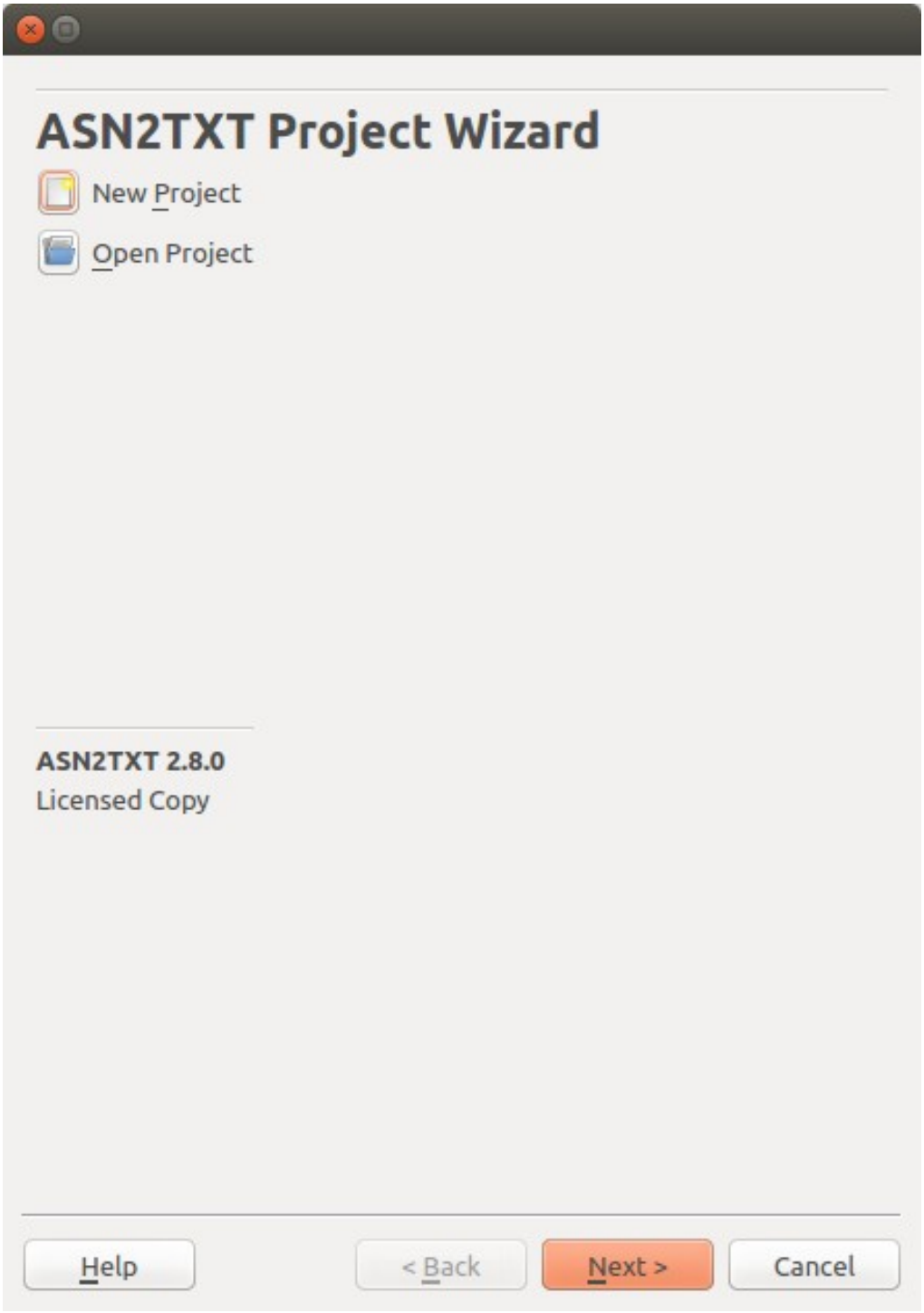
CSV Options

The following options can be used when converting to CSV.

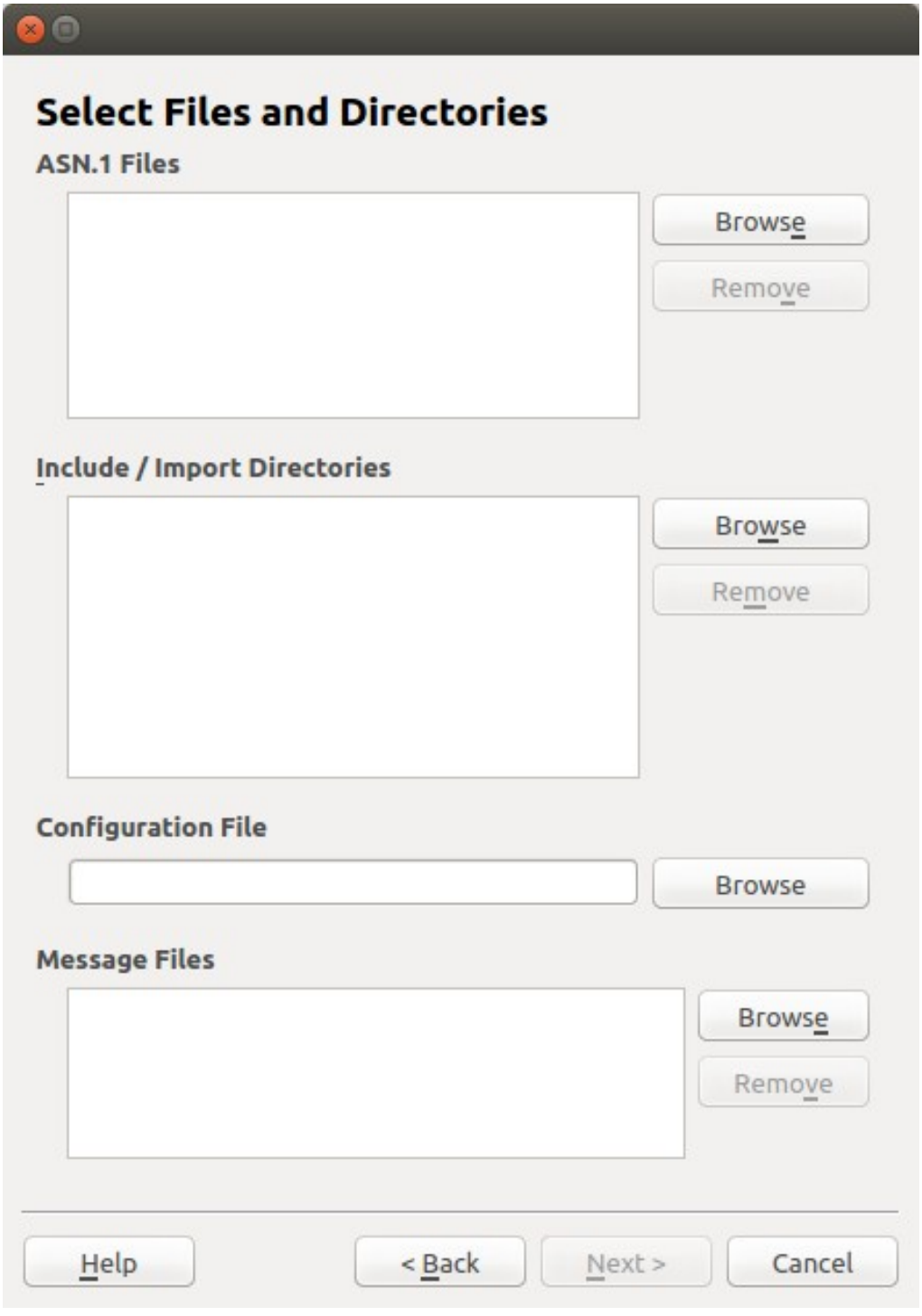
Option	Arguments	Description
-addInputFn		Add the data file name as a column to the CSV file.
-maxLevel	<level>	By default, all entries in the input file will be dumped to the output file. Deeply nested types may result in excessive output, however. The -maxLevel switch causes ASN2TXT to skip outputting data after <level> levels have been processed.
-minLevel	<level>	The -minLevel option causes ASN2TXT to unwrap top-level data types <level> levels deep.
-noquotes		By default, ASN2TXT will quote all of the fields to en-

Option	Arguments	Description
		sure that they are processed as text by spreadsheet programs to avoid converting numeric fields into scientific notation. Using this option suppresses this behavior.
-outdir	<output directory>	Specifies the directory to which ASN2TXT will output the resulting CSV files.
-padfields		Do not omit fields that would normally be duplicated in output files. Using this option will output these fields. It produces larger files, but is more explicit and may simplify ingesting the data.
-prefix		Specify the output filenames to be prefixed with the input message filename.
-s	<separator>	By default, ASN2TXT assumes the record separator will be a comma. When this conflicts with output data (e.g., a field may consist of City, State), users may use the -s switch to specify a different separator such as a tab or a pipe. Enclosing the separator in quotation marks is necessary when using a tab or other whitespace character.
-seqsep	<separator>	ASN2TXT will separate elements in a SEQUENCE OF type using a vertical bar (or pipe) by default. This option allows users to adjust the separator in case it conflicts with their expected output data format.
-separate-lines		ASN2TXT places elements in a SEQUENCE OF type on the same line if they are primitive types; this conserves space in the output file and makes it easier to process the results with text processing tools. This option ensures that all SEQUENCE OF elements regardless of type will be outputted on separate lines.

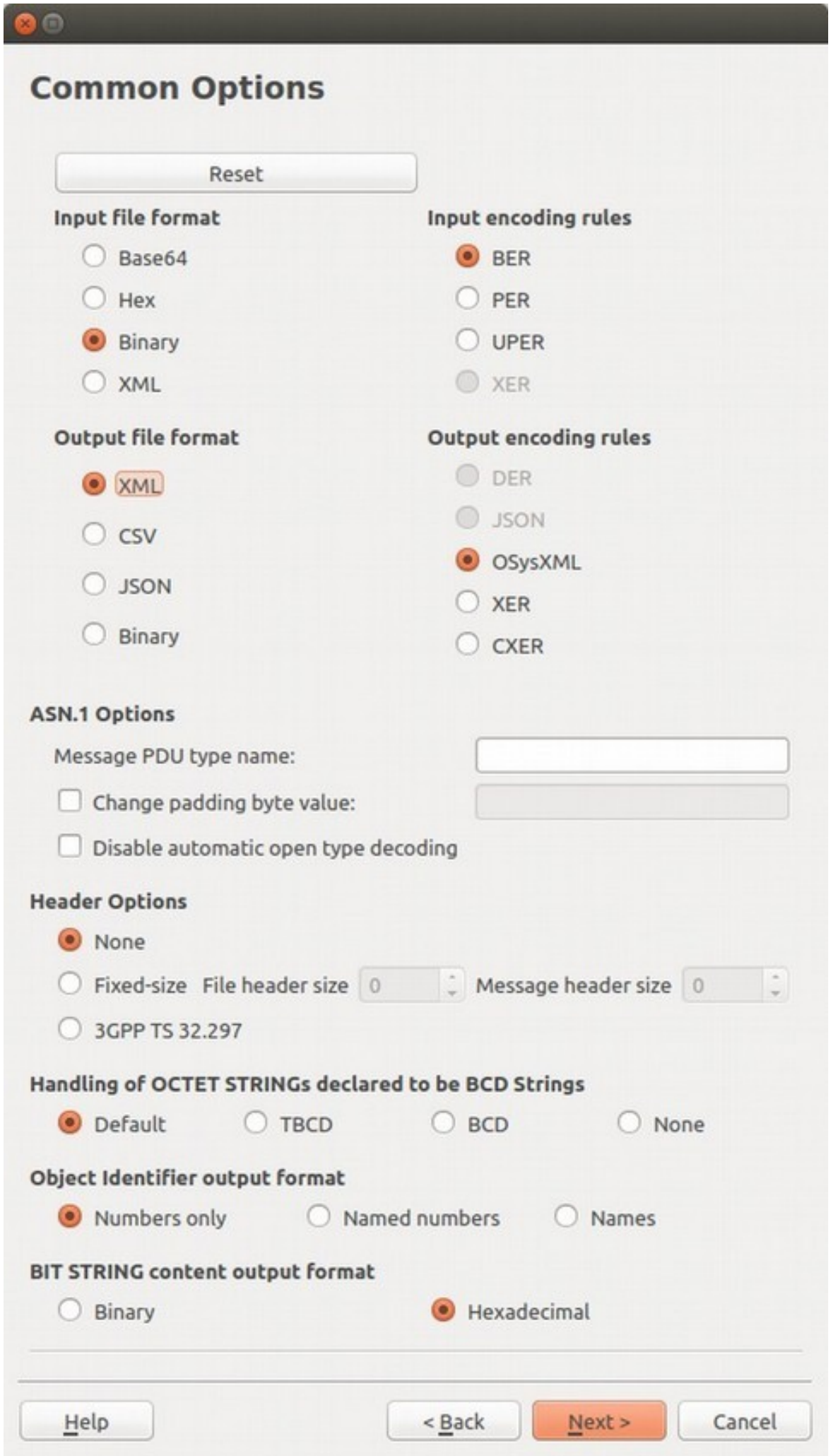
Using the GUI



ASN2TXT is equipped with a graphical user interface designed to allow users to set common options and create projects for commonly-used files and specifications. The initial screen contains options for users who wish to open or create a new project, but this is not necessary to use the software. The next screen is used to select the input specifications and message data used for decoding.

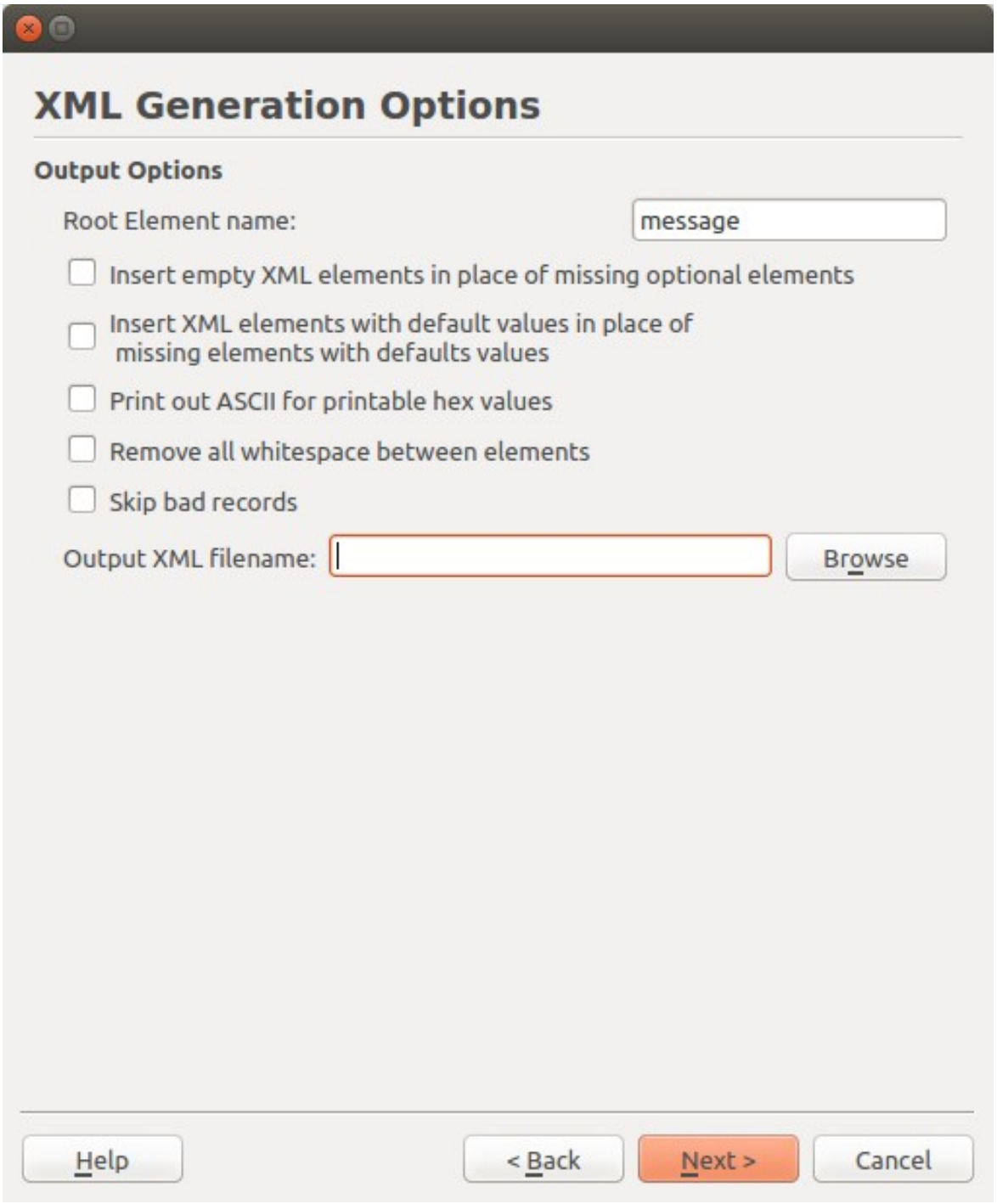


The next button in the file selection window will be inactive until the input message has been provided. If no specification is provided, CSV output will be disabled. In the next window, the following common options can be set:


Common Options
Reset
Input file format
Base64
Hex
Binary
XML
Input encoding rules
BER
PER
UPER
XER
Output file format
XML
CSV
JSON
Binary
Output encoding rules
DER
JSON
OSysXML
XER
CXER
ASN.1 Options
Message PDU type name:
Change padding byte value:
Disable automatic open type decoding
Header Options
None
Fixed-size File header size 0 Message header size 0
3GPP TS 32.297
Handling of OCTET STRINGs declared to be BCD Strings
Default TBCD BCD None
Object Identifier output format
Numbers only Named numbers Names
BIT STRING content output format
Binary Hexadecimal
Help
< Back
Next >
Cancel

Within the input and output groups of radio buttons, selecting certain options will cause some options in the other box to be disabled. The Reset button at the top will re-enable all radio buttons and will show none of them selected.

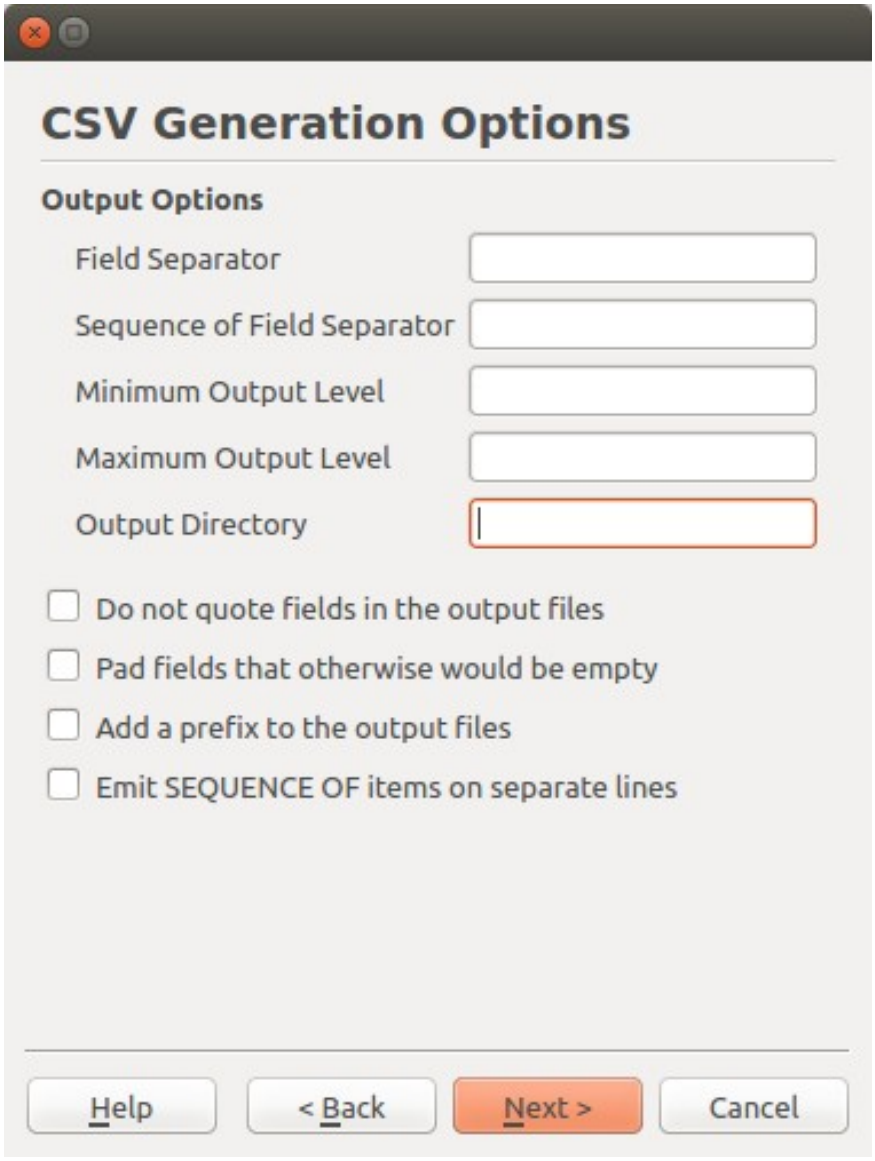
If XML is the selected output format, the following screen will appear:



Users have two options for how to direct their XML output: it can be dumped to the GUI directly or else to a file. When the XML output filename is not provided, users will see the decoded XML output in the compilation window.

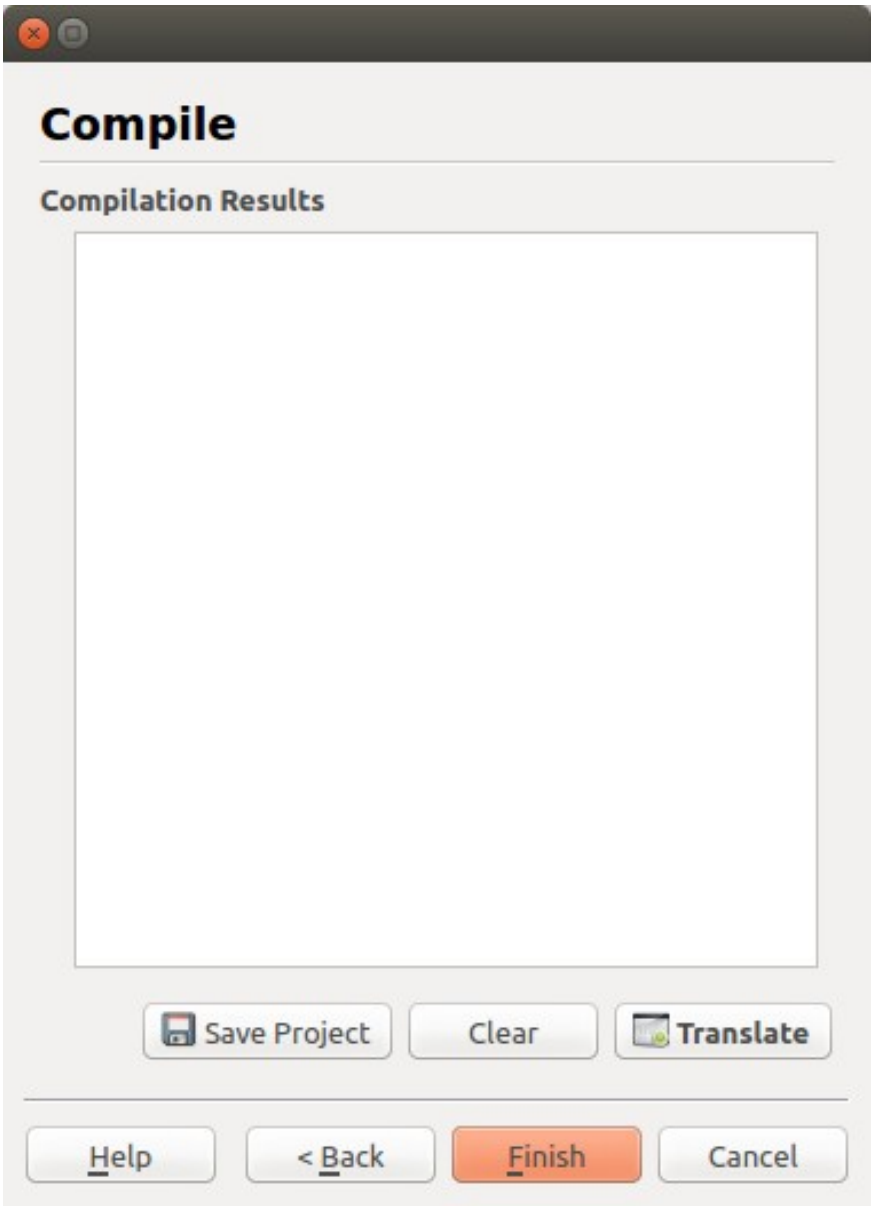
If JSON is the selected output format, it will look very similar to the above XML options screen.

If output to CSV is requested, the following screen will appear instead:



Unlike XML output, CSV output is always directed to a file (or, more likely, several files). The output in the compilation window is therefore a little different than what is seen when XML output is selected without an output filename. This behavior is normal.

When the appropriate options for either type have been selected, the following screen is presented for compilation:



Clicking the “Translate” button will invoke ASN2TXT's translator. Output may be directed to the compilation results window, to a file, or to several files depending on the choices made in previous windows.

Configuration Files

ASN2TXT provides the option of including a configuration file during the translation. This allows the user to set certain options for specific items in the schema.

Option	Values	Description
displayFormat	ipv4 | ipv6 | tbcd	Formats an OCTET STRING (“789ABCDE”, for example) as an IPv4 address (“120.154.188.224”), IPv6 address (“78:9A:BC:DE”), or TBCD string (“87*9a#cb”), respectively. “imei” and “imsi” are aliases for “tbcd.”
filter		Allows the specification of a filter that can be used to select data for output to a CSV file. See sections below for more information.
format	hex	When applied to integer types, the format option allows users to output the contents in base 16 rather than base 10. Output is prefixed with “0x” to explicitly denote hexadecimal digits.

Configuration files are formatted as XML and use the top-level tag <asn1config>. Below the top level, <module>, <production>, and <element> tags can be nested, one within the other. At each level, the tag must include the “name” attribute.

So for example, given a schema like this:

```
MyData DEFINITIONS ::= BEGIN

IPAddress ::= OCTET STRING (SIZE (4))

TwoIPAddresses ::= SEQUENCE {
      ipAddress1  IPAddress,
      ipAddress2  IPAddress
}

END
```

A configuration file might look like this:

```
<asn1config>
      <module name=”MyData”>
            <production name=”TwoIPAddresses”>
                  <element name=”ipAddress1”>
                        <displayFormat>ipv4</displayFormat>
```

```
                </element>
            </production>
        </module>
    </asn1config>
```

In this case, the configuration specifies displayFormat at the element level for the ipAddress1 element. Note that the <element> tag must be nested within a <production> tag (which must also be nested within a <module> tag). Then, whenever a TwoIPAddresses is translated, it would be output like this (in XML, for example):

```
<message>
      <!-- message 1 -->
      <TwoIPAddresses>
            <ipAddress1>123.45.67.89</ipAddress1>
            <ipAddress2>7B2D4359</ipAddress2>
      </TwoIPAddresses>
</message>
```

Since the displayFormat option was set at the element level and only for ipAddress1, only that element is affected. In order to apply such formatting for all IPAddress types, the option can be set at the production level like so:

```
<asn1config>
      <module name=”MyData”>
            <production name=”IPAddress”>
                  <displayFormat>ipv4</displayFormat>
            </production>
      </module>
</asn1config>
```

Note that in this case, the <production> tag's name attribute specifies the type which will be affected, rather than the type containing the affected element, as above.

ASN.1 Tag Path Filtering

ASN.1 tag path filtering allows elements within a BER-encoded data file to be selected for output to an XML, JSON, or Brace Text file when ASN.1 schema information is not available. A “tag path” is a concatenated list of ASN.1 tag values that target specific elements within a message. For example, in the following ASN.1 type:

```
SimpleSequence ::= SEQUENCE {
   a [0] INTEGER,
   b [1] BOOLEAN,
   c [2] SEQUENCE {
      aa INTEGER,
      bb BOOLEAN
   }
}
```

The tag path to element a would be [UNIVERSAL 16][0]. [UNIVERSAL 16] is the universal tag for SEQUENCE which occurs at the outer level. Context-specific zero ([0]) is the tag of the a element.

The tag path for the aa element would be [UNIVERSAL 16][2][UNIVERSAL 2]. In this case, aa is three levels deep so there would be three tags in the path. The final tag ([UNIVERSAL 2]) is the UNIVERSAL tag for the INTEGER data type and is used because the element is untagged.

The syntax used to express tag paths in a filter is a simplified version of the syntax used in the paths above. Instead of using the full word 'UNIVERSAL' for a universal tag, only the letter 'U' is used. There is also no space between the U and the identifier code. So the universal tag for SEQUENCE – [UNIVERSAL 16] – would be expressed in the compact notation as [U16]. For application tags, the word APPLICATION is replaced with the letter 'A' and for PRIVATE tags, the letter 'P' would be used. Context-specific tags would be as they are in the specification with no qualifying letter or preceded by the optional letter 'C'. So, for example, the two tag paths above would be expressed in compact syntax as follows:

- [U16][0]
- [U16][2][U2]

or, alternately:

- [U16][C0]

- [U16][C2][U2]

Tag paths can also contain the wildcard character '*' to indicate any tag at the given level be matched. A tag specified as [*] would match any tag at the given location. The wildcard character can also be used with a class letter to indicate any tag of a particular class be matched. For example, [C*] would match any context-specific tag.

The negation operator '~' can be used to indicate any tag except the given tag at the given level be matched. For example, [~U16] would indicate match anything except a UNIVERSAL SEQUENCE tag.

In addition to the tag path, the following items can be used in a tag path specification:

- name – a name that will be used in some types of output formats in place of the generated tag name.
- type – the data type of the element used to format the value for output. If no type is provided, the data is displayed in hex dump format. Recognized types include boolean, integer, character string, object identifier (oid), IP v4 address, and TBCD string.
- value – a textual value that would replace the actual data at the tag location. An example of its use would be to identify the type of element data in a CHOICE construct.

The specification of a complete tag path is expressed in XML as follows:

```
<asn1TagPath>
  <path> path in tag path format </path>
  <name> name of the element </name>
  <type> data type of the element </type>
  <value> value to use for the element </value>
</asn1TagPath>
```

A tag path filter consists of one or more of these elements wrapped in an <asn1TagFilter> element. These elements form a logical or of items within a message for selection. In other words, if any of the tag paths in a filter are matched, then the element is considered to be matched and is included in the output of the translation.

An example of a tag path filter that can be used to retrieve two of the elements within the employee sample program is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<asn1TagFilter>
   <Asn1TagPath>
      <path>[A0][A1]</path>
   </Asn1TagPath>
   <Asn1TagPath>
      <path>[A0][A2]</path>
      <type>integer</type>
   </Asn1TagPath>
</asn1TagFilter>
```

In this case, the APPLICATION 1 and APPLICATION 2 tagged items are selected from within the APPLICATION 0 container within the message. The type of the second item (the employee number) is specified to be integer so that the data is output in integer format. Note that the name and type elements within a tag path record are optional.

The full ASN.1 specification for an ASN.1 tag path filter is as follows:

```
Asn1TagFilter DEFINITIONS AUTOMATIC TAGS ::= BEGIN

Asn1TagPath ::= SEQUENCE {
  -- tag path in format [t1][t2]...[tn] where t is U, A, P, or empty
  path UTF8String,
  -- name to use in place of tags in printed output (for example, CSV column)
  name UTF8String OPTIONAL,
  -- type of data
  type ENUMERATED {
    boolean, integer, charstr, oid, ipv4addr, tbcdstring } OPTIONAL,
  -- Hard-coded value to use for tag match in place of value at location.
  -- This is used for CSV to indicate what type of data is at a given
  -- path (for example, what type of event in a CDR).
  value UTF8String OPTIONAL
}

Asn1TagFilter ::= SEQUENCE OF Asn1TagPath

END
```

Using the ASN2TXT DLLs

Users who wish to create their own applications for converting ASN.1 binary formats into text may use the ASN2TXT DLLs, which is included in the package as dynamic libraries (`asn2txt.dll`, `asn2txtdotnet.dll`, or `libasn2txt.so`). Supporting header files are included. The API described by these files is documented in the `doc` subdirectory of the installation, and annotated samples are provided in `sample_dll`.

Setup for use from C++

The DLL may be used from a C++ program by including the `Asn2Text.h` file and instantiating the `Asn2Text` class. The application must be linked against the DLL, and users may need to set various system variables in order to find the library. (Windows users can put the library on the system `PATH` or in the same directory as their application, and UNIX users can set the `LD_LIBRARY_PATH` environment variable.)

A small program might look like this:

```
#include “Asn2Text.h”
#include <cstdio>
using namespace std;

int main(void) {
   Asn2Text *asn2text = new Asn2Text();
   printf (“ASN2TEXT object was created.\n”);
   delete asn2text;
}
```

This program doesn't actually do anything, so it isn't very useful; but it does illustrate how the object should be created and deleted.

Cleanup in C++

The DLL is responsible for allocating memory for some of its conversion tasks, and it may be necessary to force it to release memory and reset its internal state when performing multiple operations.

This is especially pertinent for GUI-driven applications that link against the DLL, where care must be exercised to isolate translations from each other. In these cases, a call to the `cleanup` method is required:

```
asn2txt->cleanup();
```

If memory leaks are observed, users should ensure that the `cleanup` method is being properly called during the application execution.

The API documentation clearly indicates those rare circumstances where users will be expected to free memory that is allocated by the DLL.

Performing Translations

In order to perform translations, a user would need to do the following:

1. Describe the input format and encoding rules of the data to be translated. Supported input formats include binary, hexadecimal text, Base64, and XML text. Supported encoding rules include BER, DER, CER, PER, UPER, XER, and OsysXML.
2. Set ASN.1 schema files that describe encoded ASN.1 data.
3. Describe the output format and encoding rules. Output formats are binary, XML, JSON, and CSV. Output encoding rules would apply to binary formats and(BER, DER, PER and UPER) and XML Text (XER, CXER, and Osys XML). This is normally done by setting up an output stream object; but for some rules such as DER, encoding to a memory buffer can be done.
4. Invoke the translation method. Several methods with names in the form of "<something>2<something>" are available; for example. "ber2der" or "xml2der". It is also possible to invoke the "toOutputStream" method which will do the translation to the given output stream.

As an example, the following program would translate BER-encoded data to DER:

```
#include "asn2txtsrc/Asn2Text.h"
#include "rtbersrc/asn1ber.h"

int main(int argc, char** argv)
```

```
{
   Asn2Text*    pAsn2Txt = new Asn2Text();
   const char*  pSchema = "employee.asn";
   const char*  pDataFile = "message.dat";
   const char*  pOutDataFile = "message.der";
   int          stat = 0;

   // Set files and encoding rules...
   if (0 != pDataFile)
      pAsn2Txt->addAsn1DataFile(pDataFile);

   if (0 != pSchema)
      pAsn2Txt->addAsn1Schema(pSchema);

   pAsn2Txt->setOutputFile (pOutDataFile);

   stat = pAsn2Txt->ber2der();

   // Report success or failure, and clean up...
   if (stat < 0) {
      printf("ber2der failed; error = %d\n%s\n",
             stat, pAsn2Txt->getErrorText());
   }
   else {
      printf("Translation was successful\n");
      pAsn2Txt->berDump();
   }

   pAsn2Txt->cleanup();
   delete pAsn2Txt;
   return stat;
```

```
}
```

In this example, an `Asn2Text` object is first created. The addAsn1DataFile and addAsn1Schema methods are then invoked to set the input data and schemas. Multi-part schemas can be added by invoking addAsn1Schema multiple times.

The call to setOutputFile is used to specify output is to be written to the given output file,

The ber2der method is then invoked to do the translation. If successful, output will be written the specified output file. The berDump method is invoked to provide a human-readable dump of the DER-encoded data.

Setup for use from C

The DLL may be used from a C program by including the `Asn2TextFunctionsC.h` file and calling the `a2tInitContext()` function. The application must be linked against the DLL, and users may need to set various system variables in order to find the library. (Windows users can put the library on the system `PATH` or in the same directory as their application, and UNIX users can set the `LD_LIBRARY_PATH` environment variable.)

A small program might look like this:

```
#include “Asn2TextFunctionsC.h”
#include <stdio.h>

int main(void) {
   unsigned long long ullContextToken;
   ullContextToken = a2tInitContext();
   printf (“ASN2TEXT context was created.\n”);
   a2tFreeContext(ullContextToken);
}
```

This program doesn't actually do anything, so it isn't very useful; but it does illustrate how the context should be created and deleted.

Cleanup in C

The DLL is responsible for allocating memory for some of its conversion tasks, and it may be necessary to force it to release memory and reset its internal state when performing multiple operations.

This is especially pertinent for GUI-driven applications that link against the DLL, where care must be exercised to isolate translations from each other. In these cases, a call to the `cleanup` function is required:

```
a2tCleanup(ullContextToken);
```

If memory leaks are observed, users should ensure that the `cleanup` function is being properly called during the application execution.

The API documentation clearly indicates those rare circumstances where users will be expected to free memory that is allocated by the DLL.

Setup for use from .Net

The DLL may be used from a .Net program (e.g., C# or VB) by including the Com.Objsys.Asn2txt.Asn2txtdotnetdll namespace and instantiating the `Asn2Text` class. The application must be linked against asn2txtdotnet.dll, and users may need to set various system variables in order to find the library. This library is only provided with Windows kits, and Windows users can put the library on the system `PATH` or in the same directory as their application. If the latter approach is chosen, the asn2txt.dll library must also be in the same directory.

A small C# program might look like this:

```
using Com.Objsys.Asn2txt.Asn2txtdotnetdll;
using System.IO;

static void Main() {
   Asn2Text asn2text = new Asn2Text();
   Console.WriteLine (“ASN2TEXT object was created.”);
}
```

This program doesn't actually do anything, so it isn't very useful; but it does illustrate how the object should be created.

Cleanup in .Net

The DLL is responsible for allocating memory for some of its conversion tasks, and it may be necessary to force it to release memory and reset its internal state when performing multiple operations.

This is especially pertinent for GUI-driven applications that link against the DLL, where care must be exercised to isolate translations from each other. In these cases, a call to the `Cleanup` method is required:

```
asn2txt.Cleanup();
```

If memory leaks are observed, users should ensure that the `Cleanup` method is being properly called during the application execution.

The API documentation clearly indicates those rare circumstances where users will be expected to free memory that is allocated by the DLL.

Licensing

Like ASN2TXT, the DLL must have access to a valid license in order for applications linked against it to run properly. The ASN2TXT application uses an osyslic.txt license file (provided by Objective Systems) to set the license key. This osyslic.txt file can be copied into one of several places:

1. To a directory on the system-wide `PATH`.
2. To the directory indicated by the `OSLICDIR` environment variable.
3. To the directory from which the application is being executed.

The licensing API is documented in the accompanying license documentation, but of special note is the `setExeName` method. Invoking this method is only necessary when users wish to call their applications from an absolute path that differs from the current one.

We illustrate by way of a short example:

```
#include “Asn2Text.h”

int main (int argc, char **argv) {
   Asn2Text *asn2txt = new Asn2Text();
   asn2txt->setExeName (argv[0]);
```

```
}
```

The first command-line parameter (i.e., `argv[0]`) is provided to tell the license modules where they can look for an appropriate license file.

It's generally fail-safe to set the `OSLICDIR` environment variable, and users are recommended to use that method if problems arise.

Document Object Model (DOM)

A new feature of the ASN2TXT DLL is the ability to create, edit, and output an XML DOM tree. With this capability, users can add or delete nodes, or rename nodes and/or their content.

As of version 2.6, the DLL supports DOM capability to read in BER data (with or without a schema), edit it, and output the results to an XML file.

The DOM capability also allows for a DOM XML (or any XML) file to be encoded to a binary format, either BER or PER.

The sample_dll subdirectory within an installation contains a subdirectory named “dom”. Within this subdirectory are multiple examples of how to modify a DOM tree. There are example tests for adding a node, deleting a node, modifying a node, printing a DOM tree, and converting from an XML file to a binary (BER or PER) file.

Each of these examples uses command line options to pass required information to the DLL. Every example requires a data file (except for the xml_to_bin example, which requires an XML file), as well as an Xpath expression (except for the print sample). A schema option is also available (and required for the xml_to_bin example). A node name is required for the add and modify samples to specify the name of the new/modified node. See the usage() method within each example for the options available.

The doc/dll directory of an ASN2TXT installation also contains an index of html pages that outline the various new DOM-related methods available.

All of these samples are similar in how a DOM tree is set up, when converting from binary to XML. Here is a short C++ example of how to set up the DOM tree for editing (a stripped-down version of a sample; no error checking, no argument verification, etc):

```
#include "Asn2Text.h"
```

```
Asn2DomNode node;
Asn2Text*   pAsn2Txt = new Asn2Text();

pAsn2Txt->addAsn1DataFile(<path to data file>);
pAsn2Txt->addAsn1Schema(<path to schema file>);
pAsn2Txt->setInputEncRules(Asn2Text::BER);

// Initialize the DOM tree...
pAsn2Txt->toDOM();

// Print the original tree...
pAsn2Txt->print("dom_original.xml");

// Find the specified node in the DOM tree...
pAsn2Txt->findNode(<XPath expression>, NULL, &node);

// Make any changes necessary (addNode, deleteNode, etc)...

// Print the modified tree...
pAsn2Txt->print("dom_modified.xml");

pAsn2Txt->cleanup();
delete pAsn2Txt;
```

For the XML-to-binary case, the example is a little different. Much of the initial setup remains the same, but converting from XML to binary is handled with a different DLL method and there is no manipulation needed at this point:

```
#include "Asn2Text.h"

Asn2Text* pAsn2Txt = new Asn2Text();
```

```
pAsn2Txt->addAsn1DataFile(<path to XML file>);
pAsn2Txt->addAsn1Schema(<path to schema file>);
pAsn2Txt->setOutputEncRules(Asn2Text::BER);

// Convert XML file to binary format...
pAsn2Txt->toBIN(“message.dat”);

pAsn2Txt->cleanup();
delete pAsn2Txt;
```

Organizing CSV Output

As described in further sections, CSV conversions can be quite complex and verbose: complicated specifications produce lengthy column names and often overwhelm standard spreadsheet programs with column data.

To organize CSV output, ASN2TEXT has two features in particular that help generate usable data suitable for review in a spreadsheet program or for ingestion in a database. First is a pair of options, `-minLevel` and `-maxLevel`, that cause ASN2TEXT to generate separate files for some elements (`-minLevel`) and trim excess output (`-maxLevel`). Second is a new configuration option in version 2.5 for adding filters. Filters allow users to select types (and their children) for export; mapping to separate column names is also supported.

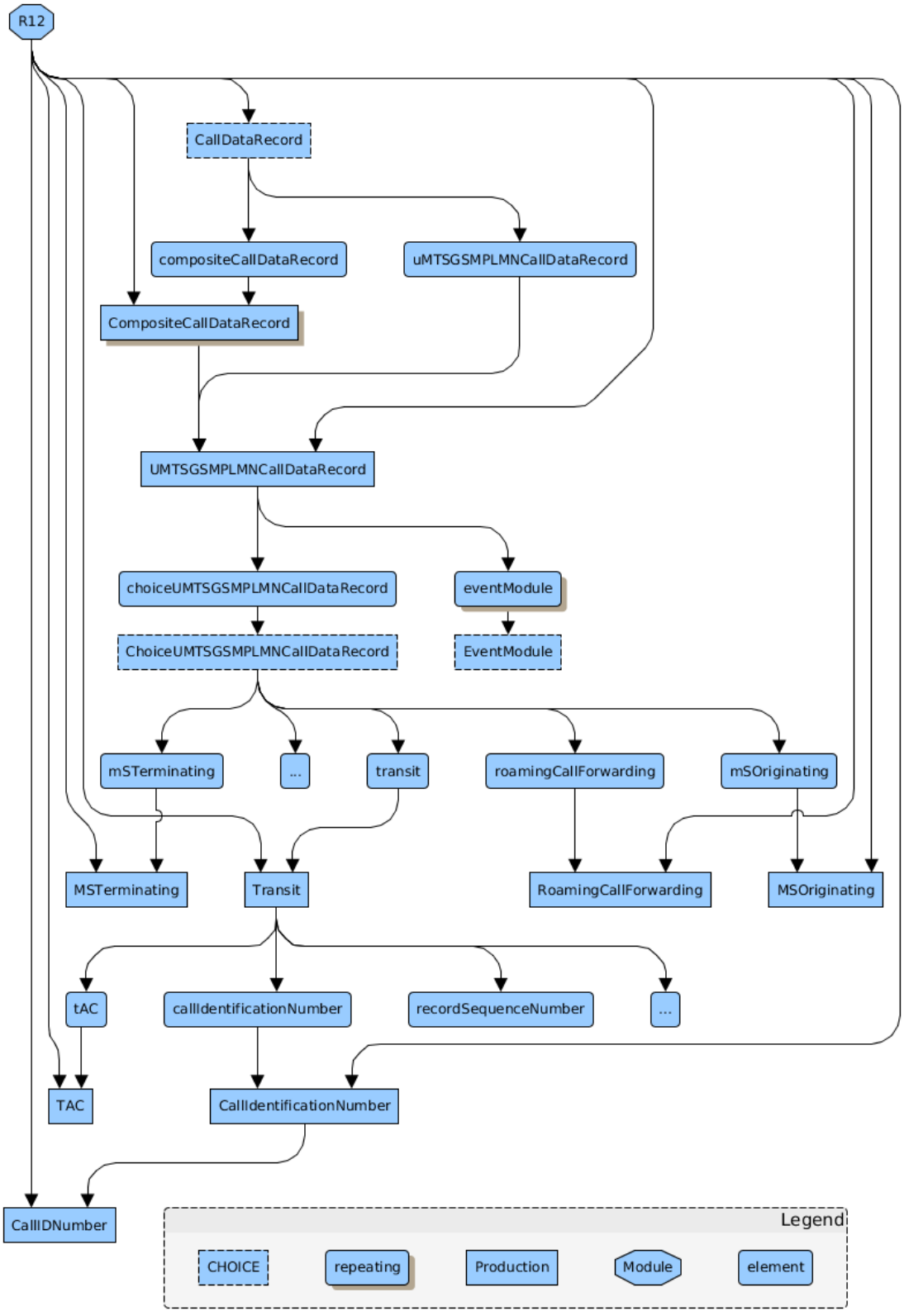
The following sections provide some examples for using these options.

Exploding and Trimming CSV Output

CSV output can often be more verbose than is needed for many applications, so users may wish to isolate or trim the CSV output using the `-minLevel` and `-maxLevel` options. Used together or separately, these two options can reorganize and truncate the CSV output. A common example in which these options are helpful with CDRs can be seen in the Ericsson R12 specification, illustrated on the next page.

In a typical run, ASN2TXT would generate a single file that contains all of the call event records and event modules: `R12_UMTSGSMPLMNCallDataRecord.csv`. Using the minimum level option, however, will allow us to “explode” the data into separate files.

ASN2TXT counts levels based on elements (illustrated in the figure as rounded rectangles). In this example, the first set of elements exists below the `CallDataRecord` type (the `compositeCallDataRecord` and `uMTSGSMPLMNCallDataRecord` elements). The second level exists below the `UMTSGSMPLMNCallDataRecord` type (the choice call data record and the event modules). It is only at the third level that we see elements that comprise the call data records of interest: `mSTerminating`, `transit`, and so on.


R12
CallDataRecord
compositeCallDataRecord
uMTSGSMPLMNCallDataRecord
CompositeCallDataRecord
UMTSGSMPLMNCallDataRecord
choiceUMTSGSMPLMNCallDataRecord
eventModule
ChoiceUMTSGSMPLMNCallDataRecord
EventModule
mSTerminating
...
transit
roamingCallForwarding
mSOriginating
MSTerminating
Transit
RoamingCallForwarding
MSOriginating
tAC
callIdentificationNumber
recordSequenceNumber
...
TAC
CallIdentificationNumber
CallIDNumber
Legend
CHOICE
repeating
Production
Module
element

Using `–minLevel 3` on the command line produces one output file for each CDR: `R12_UMTSGSMPLMNCallDataRecord_MSTerminating.csv` and others, as seen in this example execution:

```
Wrote 10 files:
R12_UMTSGSMPLMNCallDataRecord_MSTerminating.csv
R12_UMTSGSMPLMNCallDataRecord_MSOriginatingSMSinMSC.csv
R12_UMTSGSMPLMNCallDataRecord_MSTerminatingSMSinMSC.csv
R12_UMTSGSMPLMNCallDataRecord_Transit.csv
R12_UMTSGSMPLMNCallDataRecord_SSProcedure.csv
R12_UMTSGSMPLMNCallDataRecord_RoamingCallForwarding.csv
R12_UMTSGSMPLMNCallDataRecord_MSOriginating.csv
R12_UMTSGSMPLMNCallDataRecord_INIncomingCall.csv
R12_UMTSGSMPLMNCallDataRecord_INOutgoingCall.csv
R12_UMTSGSMPLMNCallDataRecord_CallForwarding.csv
```

Changing the maximum level allows users to strip off excess elements unneeded for further processing. In so doing, users can instruct ASN2TXT to isolate relevant parts of the input data file.

While it is often very difficult—if not impossible—to adequately normalize hierarchical BER data into a flat form like CSV, using the appropriate options can certainly help to facilitate transformations that are more suitable for insertion into a database.

Filtering CSV Data

New in ASN2TXT version 2.5 is the ability to specify filters within a configuration file. Filters consist of a selection *path* and an optional output *map* used to transform the column name for individual elements. ASN2TXT uses filters to match names from the input specification to decoded content. When a selection path matches, the decoded content is saved to the output CSV file; non-matching data is skipped.

This section introduces the ASN2TXT filtering syntax; we refer to the canonical employee example taken from ITU-T standard X.680 for most examples:

```
Employee DEFINITIONS ::= BEGIN
EXPORTS;

PersonnelRecord ::= [APPLICATION 0] IMPLICIT SET {
   name         Name,
   title          [0]   IA5String,
   number       EmployeeNumber,
   dateOfHire   [1] Date,
   nameOfSpouse [2] Name,
   children     [3] IMPLICIT SEQUENCE OF ChildInformation
```

```
}

ChildInformation ::= SET {
   name Name,
   dateOfBirth [0] Date
}

Name ::= [APPLICATION 1] IMPLICIT SEQUENCE {
   givenName   IA5String,
   initial     IA5String,
   familyName  IA5String
}

EmployeeNumber ::= [APPLICATION 2] IMPLICIT INTEGER

Date ::= IA5String

END
```

We begin by describing the basic filtering alphabet and then proceed to describe the configuration file syntax, filter paths, wildcards, and more complex examples.

Filter Alphabet

The filter alphabet is quite simple: all characters are treated literally except for the asterisk (*), which is used as a greedy wildcard quantifier.

Despite bearing a superficial resemblance to regular expressions, ASN2TXT filters do not treat special characters commonly used in regular expressions (such as parentheses, brackets, curly braces, etc.) any differently than normal alphabetic characters.

The filter alphabet is not restricted to valid identifiers in ASN.1 syntax, but attempting to match invalid identifiers will fail.

Filter Configuration Syntax

Filters are specified inside of the configuration file as follows:

```
<asn1config>
   <filter>
      <path>[selection]</path>
      <map>[new column name]</map>
   </filter>
</asn1config>
```

Multiple filters may be specified in a single configuration by repeating the `<filter>` element. The `<map>` item is optional; it changes the output column name for a single element.

Filter Paths

Filter paths in ASN2TXT are similar to equivalent XPath addresses applied to the XML output from ASN2TXT BER conversions. Rather than using a forward slash to separate elements, however, ASN2TXT uses a period (.) as the delimiter instead.

We use the following XML conversion of the employee example data to provide simple examples prior to describing selection rules more formally below:

```
<PersonnelRecord>
   <name>
      <givenName>John</givenName>
      <initial>P</initial>
      <familyName>Smith</familyName>
   </name>
   <number>51</number>
   <title>Director</title>
   <dateOfHire>19710917</dateOfHire>
   <nameOfSpouse>
      <givenName>Mary</givenName>
      <initial>T</initial>
      <familyName>Smith</familyName>
   </nameOfSpouse>
   <children>
      <ChildInformation>
         <name>
            <givenName>Ralph</givenName>
            <initial>T</initial>
            <familyName>Smith</familyName>
         </name>
         <dateOfBirth>19571111</dateOfBirth>
      </ChildInformation>
      <ChildInformation>
         <name>
            <givenName>Susan</givenName>
            <initial>B</initial>
            <familyName>Jones</familyName>
         </name>
         <dateOfBirth>19590717</dateOfBirth>
      </ChildInformation>
   </children>
</PersonnelRecord>
```

The path to the employee's first name is `PersonnelRecord.name.givenName`. The path to the employee number is `PersonnelRecord.number`. We can illustrate a simple selection-map combination for the whole name as follows:

```
<asn1config>
   <filter>
```

```
      <path>PersonnelRecord.name.givenName</path>
      <map>Employee First Name</map>
   </filter>
   <filter>
      <path>PersonnelRecord.name.initial</path>
      <map>Employee Middle Initial</map>
   </filter>
   <filter>
      <path>PersonnelRecord.name.familyName</path>
      <map>Employee Last Name</map>
   </filter>
</asn1config>
```

The resulting output from ASN2TXT looks like this:

```
Employee First Name,Employee Middle Initial,Employee Last Name
"John","P","Smith"
```

Informally, then, we describe the paths as `<PDU>.<element1>.<element2>...`. This is easily seen in the simple paths above: the given name of an employee can be selected using `PersonnelRecord.name.givenName`. Some complexities arise when accounting for lists of items, such as the `children` element, and a more formal definition is required.

Formally, we consider the paths to be a stack of names taken from the input specifications. As decoding occurs, names are pushed and popped from the stack. These names are usually element names, but ASN.1 also uses production names (*e.g.*, `ChildInformation`) where needed to fully qualify the decoded content.

The production name is needed in the following cases:

1. When the production is the PDU type (*e.g.*, `PersonnelRecord`).
2. When the production is a `SEQUENCE OF` type, except when it contains a `CHOICE`, a `BOOLEAN`, or an `ENUMERATED`, or when it is explicitly named (*e.g.*, if the specification were written like this:

   ```
   children ... SEQUENCE OF child ChildInformation
   ```

   the path would use `children.child` instead of `children.ChildInformation`).

The whole path can be constructed, then, by following these two principles:

1. Regardless of command-line settings, paths are addressed first using the PDU data type, *e.g.*, `PersonnelRecord`.

2. Subsequently, only element names are used unless the element is a `SEQUENCE OF` type. In this case, the element name (*e.g.*, `children`) and its production name (*e.g.*, `ChildInformation`) or child element name (*e.g.*, `child`) are both pushed.

As an example, suppose we wanted to select both the employee's name and children's names. Then we need to use a selection of this sort:

```
<filter>
   <path>PersonnelRecord.name.*</path>
</filter>
<filter>
   <path>PersonnelRecord.children.ChildInformation.name.*</path>
</filter>
```

Using this filter results in the following CSV:

```
name_givenName,name_initial,name_familyName,children_name_givenName,children_
name_initial,children_name_familyName
"John","P","Smith","Ralph","T","Smith"
,,,"Susan","B","Jones"
```

Wildcard Selections

Wildcard selections in ASN2TXT are represented by the asterisk (*), as seen above. Maps are not currently supported when using a wildcard selection.

Returning to the employee sample, we can see how to select the employee name in a single filter:

```
<asn1config>
   <filter>
      <path>PersonnelRecord.name.*</path>
   </filter>
<asn1config>
```

In this case, the output file will look like this

```
name_givenName,name_initial,name_familyName
"John","P","Smith"
```

Note that the columns revert to the ASN2TXT-selected mapping as described above.

Further Examples

The employee sample is a relatively trivial one, but we can see a significant benefit when processing large data files for relatively restricted bits of information.

Returning to the Ericsson R12 example, suppose we wanted to select only `transit` records; here are the corresponding paths:

```
CallDataRecord.compositeCallDataRecord.UMTSGSMPLMNCallDataRecord.choiceUMTSGS
MPLMNCallDataRecord.transit.*

CallDataRecord.uMTSGSMPLMNCallDataRecord.choiceUMTSGSMPLMNCallDataRecord.tran
sit.*
```

These paths capture the transit records present in both the composite call data record and the single call data record types in R12, including all of their subelements. The composite call data record is a `SEQUENCE OF` type, and therefore we must address the element's production type as well.

We find similar examples by looking at TAP3, illustrated on the following page. If we wanted to isolate the call event details, we would use a filter like this one:

```
<filter>
   <path>DataInterChange.transferBatch.callEventDetails.*</path>
</filter>
```

Combined with `–minLevel 3`, the output would be directed to one file per call event detail.

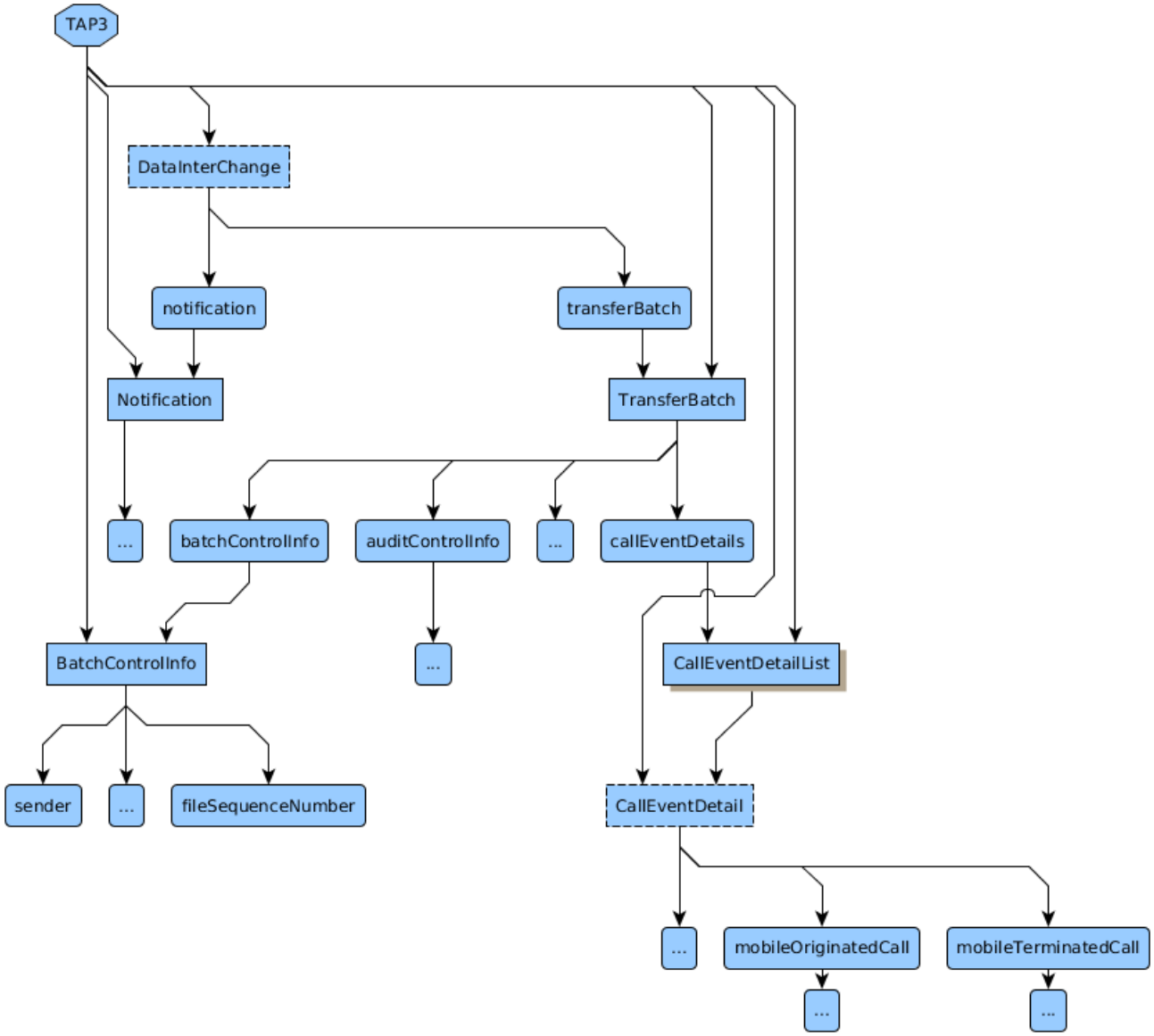
Filtering Best Practices

Avoid Using Wildcard Prefixes

The R12 filters shown above are lengthy, and could more easily be addressed through a simple filter that uses a prefix in the beginning: `*.transit.*`. This is far easier to write, but it comes with a significant performance penalty.

Matching search strings against a filter with wildcards can be computationally quite expensive because we must repeatedly return to the point of the wildcard in processing. Moreover, ASN2TXT optimizes selection using early rejection. Using a wildcard prefix harms performance by ensuring that each test will always pass. Using a wildcard prefix in local testing imposed a performance penalty of nearly 400% compared to using better-specified paths and nearly 150% compared to using no filter at all.[1]

1 R12 sample, 16 MB. A sample run using no filters processed approximately 106,000 records in 15.3s; using the longer filters to capture the transit records, processing time was cut down to 5.9s; using a wildcard prefix consumed 23.0s.



Use XML Output to Identify Paths

Identifying paths in complex specifications can be difficult. In these cases, an XML conversion can be helpful. As seen above, the XML conversion of the Employee module defined in X.680 helps to readily identify paths that can be used in the filter. In more complicated cases, this procedure is very helpful.

ASN.1 to XML Type Mappings

This chapter describes the mapping between ASN.1 encoded data values and XML for each of the ASN.1 types defined in the X.680 standard.

General Mapping without ASN.1 Schema Information

A BER, DER, or CER encoded data stream may be translated to XML format without providing associated ASN.1 schema information. In this case, XML element names are derived from built-in ASN.1 tag information contained within the message and values are encoded as either hexadecimal text, ASCII text, or as specific data-typed values if universal tag information is present.

XML element names derived from ASN.1 tag names for all tags except known universal tags is in the following general form:

```
<TagClass_TagValue>
```

where TagClass is the tag class name (APPLICATION, CONTEXT, or PRIVATE) and TagValue is the numeric tag value. For example, an [APPLICATION 1] tag would be printed as <APPLICATION_1> and a [0] tag (context-specific zero) would be printed as <CONTEXT_0>.

In the case of known universal tags, the tag value is derived using the name of the known type. In general, this is the type name defined in the ASN.1 standard with an underscore character used in place of embedded whitespace if it exists. The following table shows the XML tag names for the known types:

Tag	XML Element Name
UNIVERSAL 1	BOOLEAN
UNIVERSAL 2	INTEGER
UNIVERSAL 3	BIT_STRING
UNIVERSAL 4	OCTET_STRING
UNIVERSAL 5	NULL
UNIVERSAL 6	OBJECT_IDENTIFIER
UNIVERSAL 7	OBJECT_DESCRIPTOR

Tag	XML Element Name
UNIVERSAL 8	EXTERNAL
UNIVERSAL 9	REAL
UNIVERSAL 10	ENUMERATED
UNIVERSAL 12	EMBEDDED_PDV
UNIVERSAL 13	RELATIVE_OID
UNIVERSAL 16	SEQUENCE
UNIVERSAL 17	SET
UNIVERSAL 18-22, 25-30	Character string
UNIVERSAL 23	UTCTIME
UNIVERSAL 24	GENERALIZEDTIME

Element content will be formatted in one of three ways: hexadecimal text, ASCII (character) text, or specific-typed value.

Hexadecimal text is the default format for untyped content. ASCII text will be used if the `-ascii` command-line switch is specified and all byte values within a particular field are determined to be printable ASCII characters. A specific-type value encoding will be done if a known universal tag is found. The mapping in this case will be as described in the "Specific ASN.1 Type to XML Value Mapping" section below.

General Mapping with ASN.1 Schema Information

ASN.1 schema information is used if one or more ASN.1 schema files are specified on the command-line using the `-schema` command-line switch. In this case, element names as specified in the schema file are used for the XML element names and the content is decoded based on the specific type.

It is possible to use the `–pdu` command-line switch to force the association of a type within the specification to the message. This is only necessary if the ASN.1 files contain multiple types with the same start tag as the message type. Otherwise, the program will be able to determine on its own which

type to use by matching tags. This is true for BER/DER/CER messages only: for PER, it is necessary to specify the PDU type along with the schema.

Specific ASN.1 Type to Value Mappings

This section defines the type-to-value mapping for each of the specific ASN.1 types. By default, these mappings are not in the form defined in the ASN.1 XML Encoding Rules (XER) standard (ITU-T X.693).

When a schema is provided using the `-schema` option, the output may be adjusted to conform to XER if desired by using the `-xer` option. XER is more verbose and less validation-friendly than our native XML export. It is provided for those occasions when strict conformance is required. Differences between the two formats are provided along with the schemaless mappings below.

BOOLEAN. An ASN.1 boolean value is transformed into the keyword 'true' or 'false'. If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, a <BOOLEAN> tag is added.

b BOOLEAN ::= TRUE	
Schemaless	`<BOOLEAN>TRUE</BOOLEAN>`
XML Mode	`<b>true</b>`
XER Mode	`<b><TRUE/></b>`

INTEGER. An ASN.1 integer value is transformed into numeric text.

If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, an <INTEGER> tag is added.

i INTEGER ::= 35	
Schemaless	`<INTEGER>35</INTEGER>`
With schema	`<i>35</i>`

ENUMERATED. An ASN.1 enumerated value is transformed into the enumerated identifier text value. If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, an <ENUMERATED> tag is added.

`colors ENUMERATED {r, g, b} ::= g`	
Schemaless	`<ENUMERATED>1</ENUMERATED>`
XML Mode	`<colors>g</colors>`
XER Mode	`<colors><g/></colors>`

BIT STRING. An ASN.1 bit string value is transformed into one of three forms:

1. Binary Text (0's and 1's)
2. Hexadecimal text
3. Named bit identifiers

Binary text is the default output format. This is used if the bit string type contains no named bit identifiers and if specification of hexadecimal output was not specified on the `asn2txt` command-line.

Hexadecimal text is displayed when the `-bitsfmt hex` command-line option is used. Any unused bits in the last octet are set to zero. Note that the other bits are displayed in most-significant bit order as they appear in the string in the last byte (i.e., they are not right shifted). For example, if the last byte contains a bit string value of 1010xxxx (where x denotes an unused bit), the string is displayed as A0 in the XML output, not 0A.

Named bit identifiers are used in the case of a bit string declared with identifiers. In this case, the XML content is a space-separated list of identifier values corresponding to the bits that are set. It is assumed that bits in the string all have corresponding identifier values.

If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, a <BIT_STRING> tag is added.

bs BIT STRING {z(0), a(1), b(2), c(3)} ::= '100100'B	
Schemaless	<BIT_STRING>100100</BIT_STRING>
With Schema	<bs>100100</bs>

OCTET STRING. An ASN.1 octet string value is transformed into one of two forms:

1. Hexadecimal text
2. ASCII character text

Hexadecimal text is the default display type. ASCII text will be used for the content when the `ascii` command-line option is used and the field contains only printable ASCII characters. A special case of OCTET STRING handling is for binary-coded decimal (BCD) data types; these will be formatted as described below.

If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, a <OCTET_STRING> tag is added.

Binary-coded Decimal String. Binary-Coded Decimal (BCD) strings and Telephony Binary-Coded Decimal (TBCD) strings are not part of the ASN.1 standard, but their use is prevalent in many telephony-related ASN.1 specifications. Conversion of these types into standard numeric text strings is supported.

BCD strings usually pack two numeric digits into a single byte value by using a four-bit nibble to hold each digit. (Occasionally the nibbles are interpreted as control characters like like #, *, and so on.) By convention, the nibbles are swapped in TBCD strings, but there are no official standards for this encoding.

The `-bcdhandling` command-line option can be used to force a certain type of conversion if an encoding does not follow the usual conventions. The default handling is to reverse digits in strings determined to be TBCD strings and not reverse digits in BCD strings. The `bcd` option instructs ASN2TXT not to reverse digits for all BCD strings. The `tbcd` option instructs ASN2TXT to reverse the digits for all BCD strings.

If no processing is desired, `-bcdhandling none` can be used to instruct ASN2TXT to treat these strings as simple octet strings.

os OCTET STRING ::= '3031'H	
Schemaless	`<OCTET_STRING>3031</OCTET_STRING>`
With schema	`<os>3031</os>`
With -ascii	`<os>01</os>`

NULL. An ASN.1 null value is displayed as an empty XML element. If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, a <NULL> tag is added.

n NULL ::= NULL	
Schemaless	`<NULL/>`
XML Mode	`<n/>`
XER Mode	`<n><NULL/></n>`

OBJECT IDENTIFIER and RELATIVE OID. An ASN.1 object identifier value is mapped into space-separated list of identifiers in numeric and/or named-number format. The identifiers are enclosed in curly braces ({ }). Numeric identifiers are simply numbers. The named-number format is a textual identifier followed by the corresponding numeric identifier in parentheses. It is used in cases where the identifier can be determined from the schema or is a well known identifier as specified in the ASN.1 standard.

If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, an <OBJECT_IDENTIFIER> tag is added.

oid OBJECT IDENTFIER ::= { 1 2 840 113549 1 1 2 }	
Schemaless	`<OBJECT_IDENTIFIER>{1 2 840 113549 1 1 2} </OBJECT_IDENTI-FIER>`
With schema	`<oid>{ 1 2 840 113549 1 1 2 }</oid>`

The mapping for RELATIVE OID is the same as that for OBJECT IDENTIFIER.

Character String. An ASN.1 value of any of the known character string types is transformed into the character string text in whatever the default encoding for that type is. For example, an IA5String would contain an ASCII text value whereas a BMPString would contain a Unicode value.

If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, a tag is added which is the name of the character string type as defined in the ASN.1 standard in angle brackets. For example, the default tag for a UTF8String type would be <UTF8String>.

str UTF8String ::= “testing”	
Schemaless	`<UTF8String>testing</UTF8String>`
With schema	`<str>testing</str>`

REAL. An ASN.1 real value is transformed into numeric text in exponential number format. If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, a <REAL> tag is added.

r REAL ::= 137.035999074	
Schemaless	`<REAL>137.035999074</REAL>`
With schema	`<r>137.035999074</r>`

SEQUENCE and SET. An ASN.1 sequence value is transformed into an XML value containing an element wrapper with each of the XML element encoded values inside.

name ::= SEQUENCE { ** first UTF8String,** ** middle UTF8String OPTIONAL,** ** last UTF8String** **}**	**name Name ::= {** ** first “Joe”,** ** last “Jones”** **}**

Schemaless	`<SEQUENCE>`
	` <CONTEXT_0>`
	`  <UTF8String>Joe</UTF8String>`
	` </CONTEXT_0>`
	` <CONTEXT_2>`

	<UTF8String>Jones</UTF8String> </CONTEXT_2> </SEQUENCE>
With schema	<name> <first>Joe</first> <last>Jones</last> </name>
With `-emptyOptionals`	<name> <first>Joe</first> <middle/> <last>Jones</last> </name>

When a SET is used instead, the outer SEQUENCE tag is replaced with SET. The mappings are otherwise identical.

SEQUENCE OF / SET OF. The representation of a repeating value in XML varies depending on the type of the element value.

If the value being translated is a sequence of an atomic primitive type, the XML content is a space-separated list of values. The definition of "atomic primitive type" is any primitive type whose value may not contain embedded whitespace. This includes BOOLEAN, INTEGER, ENUMERATED, REAL, BIT STRING, and OCTET STRING values.

If the value being translated is a constructed type or if it may contain whitespace, the value is wrapped in a tag which is either the name of the encapsulating type (defined or built-in) or the SEQUENCE OF element name if this form of the type was used.

If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, a <SEQUENCE> or <SET> tag is added. That is because the tag value (hex 30 or 31) is the same for SEQUENCE OF or SET OF as it is for SEQUENCE or SET.

soi SEQUENCE OF INTEGER ::= {1, 2, 3}	
Schemaless	<SEQUENCE> <INTEGER>1</INTEGER> <INTEGER>2</INTEGER>

	<INTEGER>3</INTEGER> </SEQUENCE>
With schema	<soi> <INTEGER>1</INTEGER> <INTEGER>2</INTEGER> <INTEGER>3</INTEGER> </soi>

sos SEQUENCE OF UTF8String ::= { “test 1”, “test 2” }	
Schemaless	<SEQUENCE> <UTF8STRING>test 1</UTF8STRING> <UTF8STRING>test 2</UTF8STRING> </SEQUENCE>
With schema	<sos> <UTF8String>test 1</UTF8String> <UTF8String>test 2</UTF8String> </sos>

Name ::= SEQUENCE { first UTF8String, middle UTF8String OPTIONAL, last UTF8String }	**son SEQUENCE OF Name ::= { { first 'Joe', last 'Jones' }, { first 'John', middle 'P', last 'Smith' } }**
Schemaless	<SEQUENCE> <SEQUENCE> <UTF8STRING>Joe</UTF8STRING> <UTF8STRING>Jones</UTF8STRING> </SEQUENCE> <SEQUENCE> <UTF8STRING>John</UTF8STRING> <UTF8STRING>P</UTF8STRING> <UTF8STRING>Smith</UTF8STRING>

	</SEQUENCE>
With schema. This example shows the results with `-emptyOptionals` selected. If it were not, the first `<middle/>` element would be omitted.	<son> <Name> <first>Joe</first> <middle/> <last>Jones</last> </Name> <Name> <first>John</first> <middle>P</middle> <last>Smith</last> </Name> </son>

CHOICE. The mapping of an ASN.1 CHOICE value is the alternative element tag followed by the value translated to XML format.

PDU CHOICE ::= { a INTEGER, b OCTET STRING, s UTF8String }	**c PDU ::= { a : 42 }**
Schemaless	<INTEGER>42</INTEGER>
With schema	<PDU> <a>42 </PDU>

Open Type. The mapping of an ASN.1 open type value depends on whether the actual type used to represent the value can be determined. ASN2TXT attempts to determine the actual type using the following methods (in this order):

1. Table constraints
2. Tag lookup in all defined schema types (BER/DER/CER only)
3. Universal tag lookup (BER/DER/CER only)

If the type can be determined, an XML element tag containing the type name is first added followed by the translated content of the value.

If the type cannot be determined, the open type content is translated into hexadecimal text from of the encoded value. This will also be done if the `-noopentype` command-line switch is used.

As an example, consider the AlgorithmIdentifier type used in the AuthenticationFramework and other related security specifications:

```
AlgorithmIdentifier ::= SEQUENCE {
   algorithm ALGORITHM.&id({SupportedAlgorithms}),
   parameters ALGORITHM.&Type({SupportedAlgorithms}{@algorithm})
    OPTIONAL
}
```

In this case, the parameters element references an open type that is tied to a type value based on the value of the algorithm key. Without getting into the details of the use of the accompanying information object sets, it is known that for an algorithm value of object identifier `{ 1 2 840 113549 1 1 2 }`, the type of the parameters field is NULL (i.e. there are no associated parameters). The XML translation in this case will be the following:

```
<algorithm>{ 1 2 840 113549 1 1 2 }</algorithm>
<parameters>
   <NULL/>
</parameters>
```

ASN.1 to JSON Type Mappings

This version of ASN2TXT follows ITU-T X.697 JSON Encoding Rules (JER).

Earlier versions of ASN2TXT followed a proprietary mapping. For information on that mapping and how it differs from X.697, refer to "Javascript Object Notation (JSON) Encoding Rules for ASN.1" at www.obj-sys.com/docs/JSONEncodingRules.pdf.

ASN.1 to CSV Type Mappings

This section describes how ASN2TXT maps ASN.1 types to CSV output. Unfortunately, there exists no standard for converting ASN.1 data to CSV. BER, CER, and DER data are encoded in a hierarchical format that lends itself to translation to other hierarchical formats such as XML. CSV, on the other hand, is flat data format: there are no structured types or children, and all data in a CSV file are displayed on single lines. This complicates the translation of ASN.1 to CSV, since structured data types like SEQUENCEs can be nested to an arbitrary depth or repeated an arbitrary number of times.

The problem of converting a hierarchical data format like BER to a flat format like CSV is akin to creating and normalizing a database. In this metaphor, each CSV file represents a single table.

While these limitations make conversion a difficult problem, CSV offers some advantages over XML. CSV files are usually considerably smaller than XML, since no markup is necessary to distinguish elements. Many databases import CSV data directly into tables, so no intermediate transformations are required. CSV files can be easier to manipulate procedurally; no external XML parsers are required to read the files, and many scripting languages have built-in facilities for working with comma-delimited data.

We may divide conversion into roughly two steps: collecting the column headers; and then outputting the column data. Header information comes from parsing the input specification, while the column data is found in the actual encoded content. This documentation is primarily concerned with how the column headers are collected.

Mapping Top-Level Types

PDU data types are stored in their own CSV files, usually in the form of `ModuleName_ProductionName.csv`. There are three main top-level data types of interest:

1. SEQUENCE / SEQUENCE OF
2. SET / SET OF
3. CHOICE

The list types (SEQUENCE and SET OF) are the same as the unit types. The content is repeated when needed on multiple rows of the CSV file.

Simple types may be used as top-level data types, but in practice this is rare. Translation in this case proceeds as described in the following sections.

As an example, the following SEQUENCE would be dumped to `MyModule_Type1.csv`:

```
MyModule DEFINITIONS ::= BEGIN

Type1 ::= SEQUENCE {
   ...
}

END
```

If the input file type had two such SEQUENCEs, the resulting files would be `MyModule_Type1.csv` and `MyModule_Type2.csv`.

When a CHOICE is used as the top-level data type, the typename for the CHOICE is ignored and the files are generated using the typenames in the CHOICE. For example, the following specification would generate the same output as the one with two top-level SEQUENCEs named `Type1` and `Type2`:

```
MyModule DEFINITIONS AUTOMATIC TAGS ::= BEGIN

Type1 ::= SEQUENCE {
```

```
   ...
}

Type2 ::= SEQUENCE {
   ...
}

PDU ::= CHOICE {
   t1 Type1,
   t2 Type2
}
```

When a SEQUENCE or SET OF type is used as the top level, the underlying unit type is referenced instead. For example, the following ASN.1 specification would create the file `MyModule_Type1.csv`:

```
MyModule DEFINITIONS ::= BEGIN

Type1 ::= SEQUENCE {
   ...
}

PDU ::= SEQUENCE OF Type1

END
```

In this case, the PDU type carries no extra information for outputting the data; the contents of `Type1` are outputted on separate lines.

One of the implications of this kind of translation is that the message structure cannot be reconstructed from the output data files. A top-level data type of a CHOICE, SEQUENCE, or SEQUENCE OF may result in exactly the same output files, even though the bytes of the message may differ. Such ambiguity should not cause any problems since a specification is required for decoding the ASN.1 data.

Mapping Simple Types

Simple types in ASN.1 consist of the following:

- BOOLEAN
- INTEGER
- BIT STRING
- OCTET STRING

- NULL
- OBJECT IDENTIFIER
- REAL
- ENUMERATED
- Character strings
- RELATIVE-OID
- UTCTime
- GeneralizedTime
- GraphicString
- VisibleString
- GeneralString

Each simple type is mapped to a corresponding string representation of the input data. This is a relatively straightforward conversion. Of special note, we use the BOOLEAN values "TRUE" (for any hex octet not equal to 0x00) and "FALSE" (for any hex octet equal to 0x00). NULL values are outputted simply as "NULL."

Simple type mappings require no extra logic for output. Their textual representations are generally quite straightforward. Mapping complex types, however, is more difficult.

Mapping Complex Types

Complex types of interest include the following:

- SEQUENCE / SEQUENCE OF
- SET / SET OF
- CHOICE

Complex types are more difficult to transform than simple types. They can be self-referential and nested, which complicates transformation. CSV is a flat file format that cannot properly represent

nested types in a fixed number of columns, so care must be taken in transforming the data to ensure that it is properly represented. This process is very similar to a first-order database normalization.

CHOICE

As explained in the previous section, the CHOICE at the top level is effectively ignored: the elements of the CHOICE are used to generate the output of a file instead. In the routine case where the CHOICE is contained in another data type or stands alone, the mapping is slightly different.

Take for example the following CHOICE:

```
C ::= CHOICE {
   i INTEGER,
   b BOOLEAN,
   s UTF8String
}
```

The elements contained in the CHOICE will be used as the column names. The resulting column names from this example would look like this:

```
i,b,s
```

Simple SEQUENCEs and SETs

This section describes the transformation of SEQUENCE data types. The SET data type is analogous to the SEQUENCE. The SEQUENCE OF and SET OF types are likewise equivalent.

The only significant difference between SEQUENCE and SET is that elements may be encoded in any order in a SET. ASN2TXT will order SET elements in the order they appear in the specification. The SEQUENCEs considered in this section contain only simple types to simplify the collection of header data. More complex cases are considered in the next sections.

Take, for example, the following SEQUENCE specification:

```
S ::= SEQUENCE {
   i INTEGER ,
   s UTF8String,
   b BIT STRING
}
```

Each element of the SEQUENCE will be represented by an item in the output CSV file as follows:

```
i,s,b
```

Mapping Nested Types

When a SEQUENCE or SET contains other complex data types, it is said to be *nested*. Types may be nested to an arbitrary depth in ASN.1, so the resulting output can be extremely verbose in complex specifications. Moreover, these nested types can be repeating. The following sections describe how ASN2TXT handles nested types. A SEQUENCE is exactly the same as a SET to ASN2TXT; the two types are used interchangeably in the following sections.

SEQUENCE in a SEQUENCE

One form of nested data occurs when a SEQUENCE type contains another, as in the following example:

```
A ::= SEQUENCE {
   a INTEGER,
   b SEQUENCE { aa INTEGER, bb BOOLEAN },
   c BIT STRING
}
```

In this case, the following columns would be generated in the output CSV:

```
a,b_aa,b_bb,c
```

ASN2TXT prefixes the inner elements (`aa` and `bb`) with the name of the container element (`b`).

CHOICE in a SEQUENCE

When a CHOICE appears in a SEQUENCE, each of the elements in the CHOICE is represented in the output CSV file, even though only one will be selected in any given message.

For example, take the following specification:

```
A ::= SEQUENCE {
   a INTEGER,
   b CHOICE { aa INTEGER, bb BOOLEAN },
   c BIT STRING
}
```

The elements of the choice will be prefixed with the name of the choice in the output file, as in the following:

```
a,b_aa,b_bb,c
```

SEQUENCE OF in a SEQUENCE

The last data type to consider is the SEQUENCE OF. This is handled very much like a SEQUENCE: the SEQUENCE OF is ignored and its contents are represented for the column headers as in the following example:

```
A ::= SEQUENCE {
   a INTEGER,
   b SEQUENCE OF INTEGER,
   c BIT STRING
}
```

In this case, the columns will be straightforwardly translated:

```
a,b,c
```

It is possible that the repeated data type is not primitive, but rather complex. For example:

```
A ::= SEQUENCE {
   a INTEGER,
   b SEQUENCE OF SEQUENCE {
      aa INTEGER,
      bb BOOLEAN
   },
   c BIT STRING
}
```

As before, the inner elements (`aa` and `bb`) are prefixed with the name of the outer container (`b`):

```
a,b_aa,b_bb,c
```

Data Conversion

Having collected column headers for the output CSV, the second and final step is to output the actual data from the decoded BER message. Fortunately this is considerably more straightforward than collapsing the data structures in the specification.

The main case to consider is that in which data types are repeated: when a SEQUENCE OF is nested inside of a SEQUENCE. Some brief comments follow for other nested data types.

SEQUENCE OF in a SEQUENCE

Take for example the simple case previously seen:

```
A ::= SEQUENCE {
   a INTEGER,
   b SEQUENCE OF INTEGER,
```

```
   c BIT STRING
}
```

Let us assume for sake of argument that there are two integers in the inner SEQUENCE OF. In this case, the resulting CSV file will have two rows in addition to the header row.

The common data, columns `a` and `c`, will be represented once in the output file (unless `-padFields` is specified), while the repeated element `b` will change. For example:

```
a,b,c
1,97823789324,010010
,18927481,
```

If you have chosen to pad the fields, the output will look like this:

```
a,b,c
1,97823789324,010010
1,18927481,010010
```

While this example is very simple, it is possible to nest data types to an arbitrary depth, and the representation of columns and their data can be quite large. In pathological instances, the CSV output may be larger than the output generated by other tools like ASN2XML.

Other Nested Data Types

The other nested data types, SEQUENCE and CHOICE, are relatively trivial to convert once the columns have been assembled as described in the previous section. A single row may be used to output a message without repeating types.

The CHOICE data type bears some explanation. The following specification is the same used in the previous section:

```
A ::= SEQUENCE {
   a INTEGER,
   b CHOICE { aa INTEGER, bb BOOLEAN },
   c BIT STRING
}
```

Some example output data follows:

```
a,aa,bb,c
1,,FALSE,101010
2,137,,100001
```

The output lines will contain data in either the `aa` or `bb` columns but not both. Only the selected data should be represented in the output line.

OPTIONAL and DEFAULT Elements

Optional primitive elements that are missing in an input message will result in a blank entry in the output CSV file. Take, for example, the following specification:

```
A ::= SEQUENCE {
   a INTEGER,
   b UTF8String OPTIONAL,
   c BIT STRING
}
```

This might result in the following output:

```
a,b,c
1,test string,100100
2,,100101
3,another test,100100
```

In this example, the second message does not contain the optional UTF8String, so it is omitted from the output.

Elements marked DEFAULT are handled differently in the output. If an element is missing in the input specification, the default value is copied into the output CSV file. The following specification is used to demonstrate:

```
A ::= SEQUENCE {
   a INTEGER,
   b UTF8String DEFAULT "test",
   c BIT STRING
}
```

In this case, we might have the following output:

```
a,b,c
1,test string,100100
2,test,100101
3,another test,100100
```

Like the previous example, the input data omitted the default UTF8String. Instead of a blank entry, however, the output CSV data contains `test`.