ASN2CSV

Version 2.1.0

Objective Systems, Inc. — July 2010

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

Overview of ASN2CSV

ASN2CSV is a command-line tool that translates ASN.1 data encoded in the Basic, Canonical, or Distinguished Encoding Rules (BER/CER/DER) to a comma-separated text format (CSV) suitable for use with spreadsheet tools or databases. Unlike some tools provided by Objective Systems, ASN2CSV does not support messages encoded using the Packed Encoding Rules (PER).

ASN2CSV is envisioned primarily as a tool for working with call data records (CDRs) in a variety of formats such as TAP3, SGSN, R12, CCN, SR13, and others. It therefore does not support more advanced features of the ASN.1 standards such as two-phase decoding or information objects.

There exists no standard for converting ASN.1-encoded data to CSV. BER, CER, and DER data are encoded in a hierarchical format that lends itself to translation to similar 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 `SEQUENCE`s can be nested to an arbitrary depth or repeated an arbitrary number of times.

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 are also 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.

This document describes some of the unique challenges of transforming ASN.1-encoded data to CSV and the approach taken by ASN2CSV to solve those problems.

Using ASN2CSV

Installation

ASN2CSV 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:

```
asn2csv_v21x
|
+-asn1specs
|
+-bin
|
+-doc
|
+-sample
```

The `bin` subdirectory contains the `asn2csv` executable. The `asn1specs` directory contains specifications used by the sample programs in the `sample` directory. This document is found in the `doc` directory.

Installing on a Windows System

To install ASN2CSV on a Windows system, simply double-click the executable installer program. Selecting the default installation options will install ASN2CSV in `c:\asn2csv_v21x`.

There is no graphical user interface available for use with ASN2CSV; the program is intended to be run from the command-line, either as a stand-alone application or as part of a batch process for converting BER-encoded data to CSV.

Installing on a UNIX System

To install ASN2CSV 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 ASN2CSV to `/usr/local` or other common installation paths.

There is no graphical user interface available for use with ASN2CSV; the program is intended to be run from the command-line, either as a stand-alone application or as part of a batch process for converting BER-encoded data to CSV.

Command-line Options

Invoking `asn2csv` will show a usage message that contains the command-line options. The usage statement should look like this:

```
ASN2CSV, Version 2.1.x
ASN.1 to CSV translation tool
Copyright (c) 2004-2010 Objective Systems, Inc. All Rights Reserved.

Usage: asn2csv <filename> options

    <filename>                ASN.1 message file name

  options:
    -schema <filename>        ASN.1 definition file name(s)
    -pdu <typename>           Message PDU type name
    -o <filename>             Output XML filename
    -nobcd                       Disable BCD conversion
    -noopentype                  Disable automatic open type decoding
    -paddingbyte <hexbyte>    Additional padding byte
    -rootElement <element>    Root Element Name
    -bitsfmt <hex|bin>        BIT STRING content output format
    -inputFileType <binary|hextext|base64>
                              Format of data in input file
    -s <separator>            Field separator
    -minLevel <num>           Set the minimum output depth
    -maxLevel <num>           Set the maximum output depth
    -q                           Turn off all output except errors
```

The following table summarizes the command-line options. Required elements are listed first.

Option	Arguments	Description
<filename>		`<filename>` is the name of the input BER-encoded message data to be decoded. This element is *required*.
-schema	<filename>	This option is *required*. Us must specify a schema to apply to the input message. ASN2CSV converts the input schema items into a set of named columns and cannot name the columns without an input specification.
-bitsfmt	<hex | bin>	`-bitsfmt` may be used to 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>	`-inputFileType` may be used to tell ASN2CSV how the input data are formatted. By default ASN2CSV will assume that the input data are binary, but it can also decode hexadecimal or base64 encoded data. Whitespace in the input is ignored when `hextext` is specified.
-maxLevel	<level>	By default, all entries will be dumped to the output file. Deeply-nested types may result in excessive output, however. The `-maxLevel` switch

Option	Arguments	Description
		causes ASN2CSV to stop outputting data after <level> levels have been processed.
-minLevel	<level>	Similar to the `-maxLevel` option, the `-minLevel` option will cause ASN2CSV to skip outputting top-level data types `<level>` levels deep.
-nobcd		This option disables the conversion of BCD data types in the output. It is used for the common `TBCD-String` data type. TBCD digits are encoded in swapped byte order and use a `0xf` digit to terminate the string. When this option is selected, the input data are treated as `OCTET STRINGs`.
-noopentype		This option disables the conversion of open types in the CSV output. Typically
-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 padding, normally 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.
-q		This option causes ASN2CSV to operate in a "quiet" mode more suitable for batch processes. Informational messages are limited and only error output will be reported.
-s	<separator>	By default, ASN2CSV assumes the record separator will be a comma. When this conflicts with output data (for example, a field may consist of `City, State`), users may use the `-s` switch to specify a different separator such as a tab or pipe. Enclosing the separator in quotation marks is necessary when using a tab or other whitespace character.

Filtering Data

As explained in the following chapter, *Type Mappings and Data Conversion*, the use of nested and repeating data types can result in output files with large numbers of columns and rows. The `-minLevel` and `-maxLevel` command-line options are used to create vertical slices from an input data file.

The following example specification demonstrates how these options work to reduce the output:

```
A ::= SEQUENCE {
   a INTEGER,
   b SEQUENCE OF SEQUENCE {
      bb VisibleString,
      cc CHOICE {
         aaa INTEGER,
         bbb SEQUENCE OF BOOLEAN
      }
   }
}
```

Without using any command-line filtering options, the output columns will look like this:

```
a,bb,aaa,bbb
```

The innermost `SEQUENCE OF` type will cause a full tuple to be added to the CSV file for each message. If the `bbb` element were repeated ten times, the outer elements would be duplicated ten times for each `BOOLEAN`.

If, in the same message, the outer `SEQUENCE OF` (that is, the `b` element) were repeated three times, the outer `INTEGER`, `a`, would be repeated 30 times. This kind of duplication may be unnecessary depending on the content of interest, so the `minLevel` and `maxLevel` options may be used to control the output.

The duplication of data at the outer level may be controlled using the `minLevel` option. If for example, the minimum level were set to one (`-minLevel 1`), the outer `INTEGER` would be eliminated:

```
bb,aaa,bbb
```

The duplication of data in the inner levels may be controlled using the `maxLevel` option. If, for example, the maxLevel were set to one (`-maxLevel 1`), the inner `CHOICE` would be eliminated:

```
a,bb
```

By combining the options, we can reduce the output to a single column of data (`-minLevel 1 -maxLevel 1`):

```
bb
```

In this way the data of interest may be isolated in the input messages and the output considerably reduced.

Type Mappings and Data Conversion

Converting ASN.1 types to CSV output is not always very straightforward. It is akin to normalizing a database, except that there is only one table. For complex types, it is necessary to duplicate information across several rows.

No standards currently exist for converting ASN.1 to CSV. This chapter describes how ASN2CSV has attempted to answer the problems that naturally arise from trying to compress nested BER data to a flat data file.

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 are 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 of the form `ModuleName_ProductionName.csv`. There are three main top-level data types of interest:

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

For all intents and purposes, 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 `SEQUENCE`s, 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 `SEQUENCE`s 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
- UTF8String
- RELATIVE-OID
- NumericString
- PrintableString
- TeletexString
- VideotexString
- IA5String
- 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 by their nature 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.

CHOICEs

As explained in the previous section (*Mapping Top-level Types*), 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 name of the `CHOICE` itself will be ignored. The resulting column names from this example would look like this:

```
i,b,s
```

Basic SEQUENCEs and SETs

This section describes the transformation of `SEQUENCE` data types. The `SET` data type is analogous to the `SEQUENCE` and so bears no extra discussion. As described in previous sections, 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`. ASN2CSV 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. Other 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
```

Nested SEQUENCEs and SETs

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 will describe how ASN2CSV handles nested (and occasionally pathological) specifications. The general rule is that ASN2CSV will do its best to flatten the structure of nested data types.

For all intents and purposes, a SEQUENCE is exactly the same as a SET to ASN2CSV; 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,aa,bb,c
```

ASN2CSV removes all references to the SEQUENCE named b. Instead, the inner data (aa and bb) is collapsed into the main data type. It is as though we have instead provided the following specification:

```
A ::= SEQUENCE {
```

```
      a  INTEGER,
      aa INTEGER,
      bb BOOLEAN,
      b  BIT STRING
   }
```

While the BER encoding of the two specifications is different, they are functionally equivalent to ASN2CSV.

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 resulting columns will appear as though the `CHOICE` were actually a `SEQUENCE`:

```
a,aa,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
}
```

In this case, the innermost data are represented in the output CSV files, but the actual `SEQUENCE OF` will be ignored as before:

```
a,aa,bb,c
```

The exact same columns would be represented if a `CHOICE` were used instead of a `SEQUENCE`. In the final analysis, ASN2CSV will always do its best to collapse nested data types, drilling down to the innermost data to collect the column headers.

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 repeated, while the repeated element `b` will change. For example:

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

```
1,18927481,010010
```

The data represented by the SEQUENCE OF are different from row to row, but the common elements are duplicated. 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 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` element, 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`.