SAX-like Event Handlers for the Generic Parsing of ASN.1 Encoded Data

Objective Systems, Inc., February 2001

Abstract

The standard methodology for most ASN.1 compilers in the past has been to generate equivalent types for the different ASN.1 base types and then generate encode/decode functions to operate on variables of these types. As ASN.1 specifications become more and more complex, the need for a more generic approach becomes more apparent. The mega-type definitions that result from some of these specifications are both complicated to use and far too bulky for applications that are constrained for memory and other resources (such is the case for embedded applications).

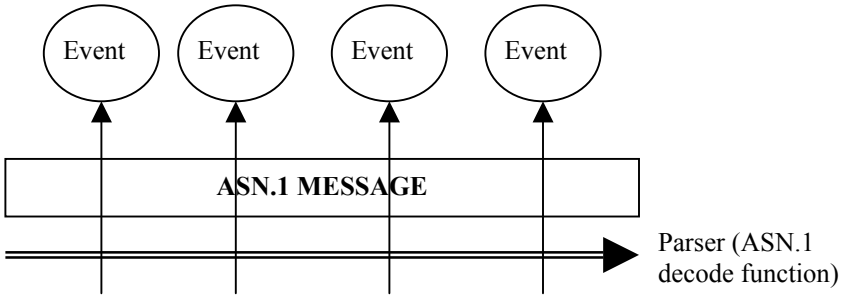
Alternatives to the standard approach can be found within the domain of the extensible markup language (XML). The Simple API for XML (SAX) and the Domain Object Model (DOM) provide alternative methods for dealing with the data without having to use type or class definitions. Together, they provide a simplified and highly object-oriented approach to dealing with XML data.

Unfortunately, in the ASN.1 world, things are not so simple. Element names are not embedded within the data so parsing engines need to couple names to events. This is where an ASN.1 compiler can help. It can parse the metadata from the ASN.1 specifications (the schemas) and include this information in the parsers that can fire events.

This paper attempts to explain the approach used by Objective Systems in the design of its next generation compiler. This compiler will provide, as an alternative to the standard type and encode/decode methods, a SAX-like parsing capability that will allow users to define event handlers to handle significant events that occur during the parsing of a message. This puts the user in control and these handlers can do whatever is required for a specific application. Examples will be shown for two applications: the formatted printing of fields with an encoded ASN.1 message and the conversion of an ASN.1 message to XML.

Introduction to SAX

Users of XML parsers are probably already quite familiar with the concepts of SAX. Significant events are defined that occur during the parsing of a message. As a parser works through a message, these events are 'fired' as they occur by invoking user defined callback functions. These callback functions are also known as event handler functions. A diagram illustrating this parsing process is as follows:



Why do this? The main reason is to put users in control of the parsing process. By allowing users to handle data as it is encountered, they can avoid having to navigate through a complex structure after parsing is complete to find a particular data item. This in turn can make parsing more efficient. Elements that are not of interest are simply discarded thereby eliminating the need for such things as dynamic memory allocation and copying that can slow down a decoder.

Furthermore, with an event-based interface, the application can start processing the data as it is parsed. With the standard compiler model, the application must wait until the message has been fully decoded before it can look at the fields.

Users can selectively filter events to look at only what they are interested in instead of having to deal with the message as a whole. Some potential applications are as follows:

- Formatted print and diagnostic handlers
- Message filtering
- Translators
- "Patch" applications in which only select fields within a message (such as timestamps) are altered.

The XML SAX model defines a large number of events that can occur during the parsing of an XML document. These events are organized into a number of Java interfaces. Events from two of these interfaces will be of interest to us: the *DocumentHandler* interface and the *ErrorHandler* interface.

Document Handler Interface

The *DocumentHandler* interface defines events that occur in the standard course of parsing a document or message. For the purposes of our implementation within ASN.1, we are only interested in three of them:

1. **startElement** – This event occurs when the parser moves into a new element. An element in XML is defined to start when a <name> tag is encountered. The name of the element is passed to the event handling callback function.

2. **endElement** – This event occurs when the parser leaves a given element space. The end of an element in XML is signaled by a </name> tag. The name of the element is once again passed to the event handling callback function.

3. **characters** – This event occurs when character data (i.e. a 'value' in name/value pairs) is encountered. This event does not necessarily have to provide the entire data value as a single event. The string can

be broken up at the discretion of the parser. It is up to the user to check for consecutive characters events and concatenate the results to get the complete value. A pointer (or reference in Java) to the character data along with a character count and offset is passed to the event handler callback function.

As an example, consider the following XML specification:

```
<employee>
  <name>John Doe</name>
  <address>1 Any Street, Anyplace, USA</address>
</employee>
```

In this specification, the following events would fire in the following order:

startElement: name = employee
startElement: name = name
characters: value = 'John Doe', count = 8, offset = 0
endElement: name = 'name'
startElement: name ='address'
characters: value = '1 Any Street, Anyplace, USA', count = 27, offset = 0
endElement: name = 'address'
endElement: name = 'employee'

The parser, at its discretion, could have broken up the characters events into separate parts. For example, there could have been multiple characters events following the address start element event as follows:

characters: value = '1 Any Street, ', count = 14, offset = 0
characters: value = 'Anyplace, USA', count = 13, offset = 14

The reasons for breaking up characters events is beyond the scope of this document. Besides, in our simplified model that will be applied to ASN.1, this will not be done. The entire value will always be passed in a single callback invocation.

Error Handler Interface

The *ErrorHandler* interface defines events that occur when errors occur in the parsing process. They allow users to take over control of the flow when an error occurs which allows for possible correction of errors in mid-stream. This inline correction can allow the parsing process to continue. Contrast this with the standard handling mechanism of throwing an exception or passing out an error status that terminates parsing. This interface allows more fault-tolerant systems to be built.

There are three methods in this interface that correspond to three levels of errors:

1. **warning –** signals problems that are not deemed serious enough to stop the parsing process.

2. **error –** signals a standard parsing error that would have resulted in a status being bubbled out of a parsing procedure. The user can perform a corrective action and return a status of ASN_OK from this procedure to allow parsing to continue.

3. **fatalError –** signals a critical error such as no dynamic memory available that would cause an immediate exit via an exit() or abort() call from the parsing application.

Applying the SAX Model to ASN.1

The application of this model to ASN.1 is not as easy as it is in the case of XML. In ASN.1, the element names are not embedded within the message. In BER/DER, there does exist a consistent nesting of

Tag/Length/Value (TLV) constructs and it would be possible to develop a tool to operate directly on the message to generate events based on the tags and contents. But the start and end element events would only be capable of returning tag values – essentially numbers as the labels identifying the data. This would not be too friendly to the human reader.

In the case of PER, not even this is possible. There are no identifiable boundaries built into the message. A message without the schema to identify it is essentially an indecipherable mass of bits.

So the ASN.1 compiler is an integral part of building a parser that can fire named events (i.e. events with meaningful element names instead of numbers). The compiler can generate decode functions that complement the usual function of populating variables of the defined types (which can be a most useful debugging aid) or which can eliminate the need for types altogether.

The model we chose to apply to ASN.1 was to keep the *startElement* and *endElement* events and to change the *characters* event. Although it would be possible to create a stringified representation of the ASN.1 values in order to implement *characters*, we felt the overhead (particularly in a C++ program) would be too high. The requirement of allocating memory for the string would mean that either the context pointer would have to be passed to the event invocation functions to allow the nibble memory allocation algorithm to be used, or direct calls to malloc or new would be needed. It seemed much easier to pass the values to the event handlers in their native form and let the user deal with converting the values to string if this was required.

So the following equivalent set of event handler callback functions was developed to replace *characters*:

boolValue Invoked when a boolean value is parsed.

intValue Invoked when a signed 32-bit integer value is parsed.

uIntValue Invoked when an unsigned 32-bit integer value is parsed (i.e. a 32-bit integer that is greater than the 32-bit integer max value).

bitStrValue Invoked when a bit string value is parsed.

octStrValue Invoked when an octet string value is parsed.

charValue Invoked when a value represented that is naturally represented as a character string is parsed. This includes the ASN.1 character string types as well as the big integer type.

nullValue Invoked when a NULL value is parsed.

oidValue Invoked when an Object Identifier value is parsed.

realValue Invoked when a real (floating point) value is parsed.

openTypeValue Invoked when an open type value is parsed.

In all cases, the argument(s) passed into the callbacks is a representation of the value that was parsed. In order to keep the interface simple, only simple value types are passed instead of structured types. So, for example, in the case of **oidValue**, the number of subidentifiers and subidentifer array are passed as separate arguments instead of as a single pointer to an ASN1OBJID structure.

The arguments passed to the *startElement* and *endElement* event handler callback functions are also slightly different to the arguments passed to the XML equivalent SAX handlers. The XML version of these functions only contains the event name as an argument. The ASN.1 version contains event name and also

has an index argument. This argument is used for SEQUENCE OF and SET OF constructs to provide the index of the element that was parsed. In all other cases, this argument is set to –1.

In terms of error handler callbacks, these are rather straight-forward and are implemented just as they would be in XML. Within a decode routine, the appropriate callback is invoked whenever a warning, error, or fatal error is encountered. The user is given access to the decode buffer object, so corrective action can be taken such as skipping an element that has problems. The user can then reset the status to ASN_OK by returning this value from the function. If the user would like the default error handling to be performed (for example, to bubble a status to the top-level on an error), all they would have to do is return the original status that was passed in.

A C++ Implementation of the Model

While it would be possible to do a C implementation of the parser, C++ is much better suited for the application due to its object-oriented capabilities. The event handler callback functions can be implemented as virtual functions within an event handler class. The event handler class would be described using an abstract base class that the user could override to provide his or her own custom functions.

The following is the abstract base class implemented within ASN1C to handle named events (note: some details have been removed to focus on the main components of the class):

```
class Asn1NamedEventHandler {
public:
   virtual void startElement (const char* name, int index) = 0;
   virtual void endElement (const char* name, int index) = 0;
   virtual void boolValue (ASN1BOOL value) = 0;
   ...
}
```

The class is called a *Named* event handler because the events reference the names of the elements within ASN.1 specification. A *Tagged* variant (Asn1TaggedEventHandler) class can also be implemented that would associate tags with values.

The start and end element methods are invoked when an element is parsed within a constructed type. The start method is invoked as soon as the tag/length is parsed in a BER message or the preamble/length parsed in a PER message. The end method is invoked after the contents of the field are processed. The *name* argument is used pass the element name. The *index* argument is used for SEQUENCE OF/SET OF constructs only. It is used to pass the index of the item in the array. This argument is set to –1 for all other constructs.

There is one value method for passing each of the ASN.1 data types. Some methods are used to handle several different types. For example, the *charValue* method is used for values of all of the different character string types (IA5String, NumericString, PrintableString, etc.) as well as for big integer values. Note that this method is overloaded. The second implementation is for 16 bit character strings which are represented as an array of unsigned short integers in ASN1C. All of the other values methods correspond to a single equivalent ASN.1 primitive type.

The user would derive his or her own class from this base class and implement the virtual methods. The user would then need to register an object of the derived class prior to calling the decode function. This is done by calling the *addEventHandler* method defined within the *Asn1DecodeBuffer* class. This registers the event handler within the context of the decode buffer object that is associated with the message to be decoded. This keep everything consistent within multi-threaded applications.

Note that event handler objects can be stacked. Several can be registered before invoking the decode function. When this is done, the entire list of event handler objects is iterated through and the appropriate event handling callback function invoked whenever a defined event is encountered.

Example 1: A Formatted Print Handler

The ASN1C evaluation and distribution kits include a sample program for doing a formatted print of parsed data. This code can be found in the *cpp/sample_per/eventHandler* directory. Parts of the code will be reproduced here for reference, but refer to the directory above to see the full implementation.

The format for the printout will be simple. Each element name will be printed followed by an equal sign (=) and an open brace ({) and newline. The value will then be printed followed by another newline. Finally, a closing brace (}) followed by another newline will terminate the printing of the element. An indentation count will be maintained to allow for a properly indented printout.

In *startElement*, we print the name, equal sign, and opening brace as follows:

```
void PrintHandler::startElement (const char* name, int index)
{
   indent();
   printf ("%s = {\n", name);
   mIndentLevel++;
}
```

In this simplified implementation, we simply indent (this is another private method within the class) and print out the name, equal sign, and opening brace. We then increment the indent level. Note that this is a highly simplified form. We don't even bother to check if the index argument is greater than or equal to zero. This would determine if a '[x]' should be appended to the element name. In the sample program that is included with the compiler distribution, the implementation is complete.

In *endElement*, we simply terminate our brace block as follows:

```
void PrintHandler::endElement (const char* name, int index)
{
   mIndentLevel--;
   indent();
   printf ("}\n");
}
```

All that each of the various value methods have to do is print a stringified representation of the value out to stdout. For example, the *intValue* callback would just print an integer value:

```
void PrintHandler::intValue (int value)
{
   indent();
   printf ("%d\n", value);
}
```

So, for example, if we have a production like this:

```
employeeNumber INTEGER ::= 55
```

We would get the following output:

```
employeeNumber = {
```

```
   55
}
```

Possibly a bit verbose for some people's needs, but it illustrates a simple way of formatting printed output. The way you format your output is up to you.

Example 2: ASN.1 to XML

Without a great deal more effort, the example above can be modified to provide a transformation of an ASN.1 message into XML. Once in this format, the output can be examined using a variety of tools and browsers now available for XML.

The *startElement* method would be modified as follows to create an XML start element block:

```
void Asn1ToXMLHandler::startElement (const char* name, int index)
{
   indent();
   printf ("<%s>\n", name);
   mIndentLevel++;
}
```

Once again, some sort of processing of the index argument might be desirable. In this case, you may want to append an "_n" to the element name to make it unique (n would be replaced by the index value).

The *endElement* method would be as follows:

```
void Asn1ToXMLHandler::endElement (const char* name, int index)
{
   mIndentLevel--;
   indent();
   printf ("</%s>\n", name);
}
```

The value methods would be implemented as before. A stringified representation of the value would be written to the output file or stdout.

This would result in the following output for the example above:

```
<employeeNumber>
   55
</employeeNumber>
```

References

(1) *XML by Example*, Benoit Marchal, Que, 2000