Experience with the Abstract Syntax Notation One and the Basic Encoding Rules

James D. Harvey and Alfred C. Weaver
Department of Computer Science
Thornton Hall
University of Virginia
Charlottesville, Virginia 22903

Abstract

This paper discusses our experience in implementing an Abstract Syntax Notation One compiler. ASN.1 is a descriptive notation used to define the informational content of the OSI upper layer protocols. The Basic Encoding Rules are a transfer syntax that may be applied to instances of ASN.1. Collectively, these two protocols solve the disparate data representation problem within an OSI network. We describe the purpose and basic features of both ASN.1 and BER, their theoretical disposition within the OSI Reference Model, their practical application that includes the details of our implementation, and finally we offer some insights into the protocols themselves.

1 Concepts

The most fundamental goal of the OSI Reference Model is to enable the interconnection of computers of different vendors, architectures and ages. This objective is clearly ambitious considering that the development of computer science has produced a vast number of machine architectures, programming languages and compiler technologies. The cross-product of these three characteristics defines a broad spectrum of data representation methodologies. The respective uniqueness of each of these methodologies represents an obstacle in a computer network. Interoperability within a heterogeneous network requires data representation commonality. It is the purpose of the Abstract Syntax Notation One (ASN.1) [1] and the Basic Encoding Rules (BER) [2] to provide a means to achieve this commonality.

Within the OSI environment the Presentation layer [3] [4] has the responsibility of solving this problem. It offers a context negotiation capability that enables it to establish a common set of data representation techniques with its peer entity. Each technique, or defined context, is identified by a pair of values, an abstract syntax name and a transfer syntax name. These agreed upon techniques form a defined-context-set. Once established, the Presentation layer can transform the local syntax of its service user's data to and from these agreed upon methodologies.

Figure 1. The Presentation Layer

To the Application layer and above, the fact that an interconnected host may not share the same method of representing data is transparent. Therefore, as user data passes through the Presentation layer its complexion changes significantly. To the layers beneath Presentation the user data parameters of a service primitive appear as a mere sequence of octets. These octets are left uninterpreted by the lower layer protocol entities that carry them. Within the Application layer the user data parameters typically correspond to complex data types that possess significant meaning to
the protocols that use them. Application entities must process the content of these values in detail to provide a requested service. The nature of this transformation of the user data can be viewed as the bridge between syntax and semantics. It is the job of Presentation to create this bridge as shown in Figure 1.

Two objectives of this task make it a difficult one. First, the Presentation layer should provide this service in such a way that it does not overly determine the manner in which the data is represented within the Application layer. Second, the definition of the user data parameter's informational content should not overly determine the syntax of the octets used to convey it. This decoupling of semantics and syntax is a most crucial aspect of how ASN.1 and BER solve the data heterogeneity problem.

ASN.1 is a notation that formalizes the data types and data values of a protocol for the purpose of standardization. Specifically, ASN.1 describes the informational content of both the user data parameters and the control information that cross the interface between Presentation and Application. The most important aspect of how ASN.1 presents this description is that it does so without dictating the representation of this data on either side of the interface. An instance of an ASN.1 definition is called an abstract syntax.

To understand the abstract syntax concept it is useful to consider the term literally. Formally, a syntax defines the sentential forms of a language. Within the context of communications, one usually thinks of a syntax as defining the bit patterns used to represent data flowing across a medium. An abstract syntax, therefore, does not define these bit patterns, but rather it establishes a framework upon which these bit patterns can be created.

The data items that are passed from Application to Presentation will become the user data parameters of the Presentation protocol. These values along with additional Presentation protocol information will in turn cross the interface between Session and Presentation as the user data parameters of the Session protocol. However, before they cross this latter boundary these data values must be transformed into an uninterpreted sequence of octets. This transformation, or encoding, of the data values of an abstract syntax into a sequence of octets is achieved through the application of a transfer syntax. The necessary properties of any transfer syntax are that it carry the information described by the abstract syntax in a manner which is independent of the representation of the data in any particular application entity, and that the reciprocal transformation, or decoding, of the octets into a local syntax be readily applicable. BER is a transfer syntax.

2 Abstract Syntax Notation One

ASN.1 is a language used to describe the data types and data values of a communications protocol. It is typically used by the OSI upper layer protocols (excluding session) to define the types of their Protocol Data Units (PDUs). ASN.1 is similar to most traditional programming languages except that it does not provide executable statements, only declarative ones. It offers a broad spectrum of fundamental types called simple types. This includes integers, booleans, enumerated types, reals, bit strings and character strings. ASN.1 also provides structured types which may be applied to these simple types and other structured types to form new types of arbitrary complexity. These structured constructs include sequence types, set types, set of types, sequence of types, choice types and selection types.

ASN.1 includes a value notation that enables a protocol designer to document specific values of interest. Types and values may be logically grouped into a single protocol definition called a module. Separate module definitions may share definitions through the use of external reference mechanisms. As such, ASN.1 not only provides the ability to define an individual protocol, but also it provides the ability to standardize the type and value interdependencies that exist between these protocols. A tutorial on ASN.1 is provided in [6].

3 Basic Encoding Rules

The Basic Encoding Rules define a technique for encoding data values corresponding to ASN.1 type definitions. A BER encoding is comprised of a sequence of octets. These octets are partitioned into a 3-tuple, depicted in Figure 2.

<table>
<thead>
<tr>
<th>IDENTIFIER</th>
<th>LENGTH</th>
<th>CONTENTS</th>
</tr>
</thead>
</table>

Figure 2. BER Format

The identifier octet(s) contain information regarding the type and form of the encoded data value. A primitive form indicates that the contents octets contain a direct representation of the data value. A constructed form indicates that the contents octets contain another embedded type-length-value (TLV) encoding. The
length octet(s) determine the end of an encoding. They indicate how many contents octets represent the encoded data or, if this number is unknown, then they indicate that the contents octets are delimited by a reserved bit pattern called an end of contents (EOC) sequence. The later case is referred to as the indefinite length form. And finally, the contents octets contain a direct representation of the data value or another nested TLV sequence.

4 Compiler

Our ASN.1 compiler is written in the C programming language and is approximately 6500 lines in length. Its front end was developed using the Unix utilities Lex and Yacc. The compiler accepts an ASN.1 module definition as input and generates two C files as output: a header file containing the C declarations of the PDUs and a code file containing encoding and decoding functions. A run-time library provides a melange of general purpose routines, most of which perform the actual encoding and decoding of the primitive types. Unfortunately the compiler does not fully support the ASN.1 standard - macro notation, useful types, subtypes and reals are not presently implemented.

To apply the BER transfer syntax the ASN.1 compiler requires precise knowledge of the structure of the user's local syntax. It is therefore appropriate and necessary that the compiler define this local syntax on the user's behalf. This provides a more robust implementation since it ensures that the ASN.1 type definitions will always be consistent with the actual PDU representations within the application.

Each type assignment within an ASN.1 module definition produces a corresponding C typedef declaration. These typedefs are subsequently embedded within a discriminated C union to represent the entire collection of PDUs. An enumerated discriminate value identifies which alternative is active. Hence, this single C declaration represents the user's local syntax which is passed to and from the generated encoding and decoding functions.

A single encoding function is generated that accepts a decoding argument as input and produces an encoding argument as output; it also returns an integer error code. The encoding process occurs in two phases. The purpose of the first phase is to calculate all of the length octet values and allocate an encoding buffer of the exact size required. This task is performed by examining the contents of each element within the PDU and making the appropriate run-time calls. These computed lengths are inserted into a generated table that also contains the pre-computed type octet values. These length values are subsequently extracted from this table within the second phase as the actual encoding is generated.

Like the encoding process, a single function is generated by the compiler to provide the decoding capability. It accepts an encoding argument as input and produces a decoding argument, whose type is the discriminated union, as output. By using this combination of a single function and a discriminated union, the caller is not required to have apriori knowledge of the PDU type contained within the encoding.

The decoding process is controlled by a finite state machine, implemented as a state transition table. Each entry of the table contains an expected tag value and a set of three transitions, M, D and E. The M transition is taken whenever the current tag value within the encoding matches the tag of the current state. The D transition is taken whenever these two values differ. Finally, the E transition is necessary when the value being decoded requires that a set of states be executed an undeterminable number of times; this is the case when the value is a set type, a set-of type, a sequence-of type or a recursive type. The actual decoding process consists of a large switch statement that is iteratively executed until either the decoding is successfully completed, or the decoding fails.

5 Observations

Through the process of building our compiler many insights into ASN.1 and BER were obtained. The most revealing pertains to the comparative computational complexities of sets and sequences.

5.1 Sets vs. Sequences

There is an efficiency consideration associated with the use of ASN.1 set types that is subtle, yet significant. Compared to the sequence type, decoding an ASN.1 set is a computationally intensive process that may represent an unnecessary protocol inefficiency. The reason for this stems from the fact that the order of elements within a set value is not significant, which increases the complexity of the decoding process considerably. Consequently, there are many instances when the use of a sequence type is preferable to the use of a set type. However, as presented in [8], this difference in efficiency is not readily apparent in ISO 8824. Therefore, it is likely that protocol designers will make design decisions regarding the use of sequence types versus set types that are not optimal.
The only difference between the ASN.1 sequence type and the ASN.1 set type is that the order of elements within the set is not significant. To the ASN.1 writer opting to define a protocol data unit as a set rather than a sequence appears to be less restrictive and somewhat arbitrary: it offers the encoding implementation the flexibility to reorder the elements during the application of a transfer syntax. Yet, regarding the implementation, the cost of this flexibility is quite high. Indeed, the decoding of a set value is far less efficient than that of a sequence due to this unpredictable ordering and, ironically, it seems unlikely that the order of the elements within an encoding of a set type will ever differ from the order of their original definition.

To comparatively analyze the computational intensity associated with decoding a set type versus a sequence type, it is necessary to view each from a common perspective. Consider the ASN.1 definition in Figure 3:

```
Set-vs-Seq DEFINITIONS ::= BEGIN
  Set-type ::= SET {
    IASSString, INTEGER, BOOLEAN };
  Seq-type ::= SEQUENCE {
    IASSString, INTEGER, BOOLEAN };
END
```

Figure 3. Sets vs. Sequences

By modeling the decoding process of both of the above types as a finite state machine (fsm), the difference in implementation efficiency can be clearly seen. The transition diagram of the Seq-type decoding fsm is shown below in Figure 4.

```
Figure 4. Sequence fsm
```

Each state of the fsm represents the universal class number that is currently being sought. There are two types of transitions, M and D. The M transition is taken whenever the current tag within the encoding matches that of the state. The D transition is taken whenever these two tag values differ. Note that the relationship between the number of elements of Seq-type and the number of states within the corresponding fsm is linear.

Now compare this with the state transition diagram that represents the decoding of the Set-type definition, Figure 5. In this second fsm each state not only represents the tag value that is currently being sought, but also a record of what elements have been decoded thus far. The tag value above the line represents the former while the set notation on the bottom represents the latter. In comparison to the Seq-type fsm, the Set-type fsm contains nine additional states. These "extra" states are necessary because the decoding logic can not assume the order of the elements within an encoding. Consequently, this memory of the previous elements must be contained within the fsm.

```
Figure 5. Set fsm
```

1 The failure state and all of the associated D transitions have been omitted to reduce the complexity of the diagram.
The complexity of the Set-type fsm can be mathematically determined using combinatorial analysis and the binomial theorem. Note that the state transition diagram in Figure 5 is partitioned into three levels, represented by the variable \( i \) at the bottom. In all three levels the number of states relative to the tag value currently being sought (the number above the line) is equal to the number of possible ways in which one object can be selected from \( n \) objects, where \( n \) is the number of elements within the set; that is, the number of states relative to the tag value is given by

\[
\frac{n!}{(n-1)!i!} = \binom{n}{i}
\]

Relative to the number of elements decoded thus far (the set notation below the line), it is clear that the total number of states at each level \( i \) is equal to the number of ways that \( i-1 \) objects can be selected from a group of \( n-1 \) objects. Specifically, the number of states at level \( i \) is equal to

\[
\binom{n-1}{i-1}
\]

Therefore, the total number of states within the Set-type fsm is equal to the summation:

\[
\sum_{i=1}^{n} \binom{n}{i} \binom{n-1}{i-1}
\]

which is also equal to

\[
n \sum_{i=0}^{n-1} \binom{n-1}{i}
\]

From the binomial theorem\(^2\) we know that

\[
2^n = \sum_{i=0}^{n} \binom{n}{i}
\]

Thus, the total number of states within the decoding fsm for a set type is characterized by the function \( n2^{n-1} \).

The higher complexity of the set fsm implies that the process of decoding a set type is more expensive than a sequence type. This additional cost will be in the form of processing time, memory usage, or both, depending upon how the fsm is implemented. Moreover, the existence of optional elements or default values within a set type magnifies this difference even further. Within a set type the processing of both of these constructs must be deferred until the entire set has been decoded. This is the only point when the presence or absence of an element is known, whereas in a sequence type these constructs may be applied as soon as the position of the element within the encoding has been reached.

5.2 Library Management

The unrestricted manner in which type definitions and value definitions from other modules may be referenced suggests that the concept of library management within the ASN.1 standard is inadequately defined. When an ASN.1 writer wishes to reference a type or value that is defined within a separate module, there are two mechanisms available, an external reference or an imported symbol. Both methods require an explicit reference to the name of the module containing the definition.

Yet, because ASN.1 does not impose restrictions upon symbolic importing, mutually dependent modules may be defined. Consider the module definitions within Figure 6.

```
Module-A DEFINITIONS ::= BEGIN
  A1 ::= BIT STRING
  A2 ::= SEQUENCE {
    Module-B.B1
  }
END

Module-B DEFINITIONS ::= BEGIN
  B1 ::= BIT STRING
  B2 ::= SEQUENCE {
    Module-A.A1
  }
END
```

Figure 6. Mutually Dependent Modules

In this example Module-A and Module-B are mutually dependent since each references a data type that is defined within the other. This potential for mutual dependencies requires that an ASN.1 compiler generate partially complete intermediate code files and makes the processing of these definitions extremely difficult.

\(^2\) let \( x^y = \sum_{i=0}^{n} \binom{n}{i} x^y \)
This problem could be avoided if ASN.1 were to establish a requirement of linear elaboration, in the same manner that the Ada programming language [5] defines the concept. In Ada, *elaboration* is defined as the process by which an entity is brought into existence. It marks the point in time when the name of the declarative item is formally bound to its type. With the exception of incomplete type specifications, the name of an Ada entity may not be used before the declarative item that declares it has been fully elaborated. From this requirement the concept of linear elaboration evolves. With respect to compilation units, linear elaboration requires that a non-circular ordering of all of the compilation units referenced, either directly or indirectly, by a given program must exist. If no such ordering can be found, then the program is erroneous.

5.3 Useful Definitions

Section three of ISO 8824 defines a category of ASN.1 types, the Useful Definitions, that are to be considered of general utility and therefore useful in a number of applications. These definitions appear to form the beginnings of a predefined library of general purpose types that is expected to grow as the use of ASN.1 matures. However, the manner in which they are incorporated within ISO 8824 makes their disposition vague and may represent a symbolic scoping inconsistency.

All of the ASN.1 Useful Definitions are defined using existing ASN.1 types and constructors. These definitions may be viewed as a predefined extension of the ASN.1 language that does not require additional support from the Basic Encoding Rules. Yet, the manner in which these definitions are packaged and presented within ISO 8824, and the manner in which a user may reference these symbols, does not coincide with this perspective.

As an extension to the ASN.1 language, it is natural to view the Useful Definitions as a separate, predefined, registered abstract syntax whose symbols are exported to the outside world. In this way, ASN.1 writers could reference these useful definitions by simply using the same external referencing mechanisms that are currently defined within the standard. This would present these Useful Definitions in a clear and consistent way.

As the Useful Definitions are currently defined, a user may reference the types without using the standard external referencing mechanisms. This implies that these definitions are part of the ASN.1 language and not an extension. As the use of ASN.1 increases so will the need for more Useful Definitions. It is therefore likely that this current disposition of the Useful Definitions will serve as an impetus for continual changes to the ASN.1 language itself. Albeit that the introduction of these new Useful Definitions will be necessary and inevitable, it is not necessary to change the language in the theoretical sense. It is important, for the sake of implementation, that ISO 8824 draw this distinction between language extension and language modification.

It is useful to note the strong similarity that exists between the ASN.1 Useful Definitions and the package STANDARD in the Ada programming language. In Ada, the distinction between what is a part of the language and what is merely an extension is clear. In all implementations of the Ada language, a programmer may not reference the predefined objects in package STANDARD without first creating a library that contains the symbolic information corresponding to this compilation unit.

From an implementation perspective it is appropriate to take this same approach with ASN.1, as discussed in [7]. Consequently, to support the Useful Definitions in an implementation, it is not necessary to add additional logic to an ASN.1 compiler. The simple requirement of pre-compiling this module will facilitate this capability.

5.4 Integers

The BER standard, ISO 8825, requires that all integer values be encoded using the minimum number of octets. This requirement prohibits integer encodings from being unnecessarily large and prevents the pathological case where an integer value is transferred with an arbitrary number of leading octets that contain all zeros or all ones. Unfortunately, the cost of this decision appears to be high with respect to efficiency. It has been shown [9] that the encoding and decoding of an BER integer value, in contrast to a simple memory copying technique, decreases the transfer rate by a factor that ranges from 5 to 20 depending upon the host system. Furthermore, in this same set of experiments it was shown that a fixed length approach reduced the encoding time by a factor of 5-6.

To support this requirement of integer encoding minimality an implementation must perform a series of checks to determine the size of each integer value. This can be accomplished through successive range checks or successive logical comparison and shift operations on the most significant octet. While the later method appears to be more efficient, it is still unnecessarily slow. Requiring integers to be encoding
in a minimum number of octets is ill-advised with respect to performance.

5.5 Object Identifiers

The encoding and decoding of object identifiers is an unnecessarily inefficient process, or at the very least it is overly complex. Comprised of a series of numeric values called components, an object identifier is encoded by representing each component as a series of seven-bit quantities. These seven-bit quantities are concatenated to form each component value. The leading bit of each contents octet is used to demarcate the boundaries between the individual components. To further complicate this algorithm the first two components are handled as a special case, packed into one octet using the equation 40x + y (where x represents the value of the first component and y represents the value of the second). This packing takes advantage of the fact that there are only three values allocated from the root of the Object Identifier Tree, and at most 39 subsequent values from nodes x=0 and x=1.

The decision to use seven-bit quantities is questionable since bit masking or arithmetic shifting is necessary to isolate the value. Furthermore, it is not clear whether a savings of one octet warrants the computationally expensive combination of the first two component values. This requires that the run-time encoding/decoding routines have built-in knowledge of the structure of the Object Identifier Tree. The application of this knowledge is expensive.

There is, however, one fact that we encountered that mitigates this inefficiency - object identifier values are most often used in an autonomous form for comparative purposes only. Consequently, it is acceptable not to decode the individual component values at all, but instead return only their address and length. If these values are always declared as encoded constants, then the comparative purposes will be met while the inefficiency of encoding and decoding will never be encountered.

5.6 Tagged Types

From a theoretical perspective the concept of explicit tags seems erroneous; it thrusts the responsibility of preventing ambiguity upon the user. It is certainly possible to establish a standard, canonical ordering of the type definitions within any module and to determine tag values from this ordering. Consequently, a compiler could potentially generate these tags and assume this responsibility instead of the user. As long as the generated tags are predictable, economical and unique, the user's needs would be met.

If such a tag generation algorithm could be defined, tag values might not be reusable in the case of mutually exclusive contexts. Therefore, the point at which extended tag values are necessary might arrive sooner than before and would cause the average length of an encoding to increase. However, this point would depend upon the extent of the mutually exclusive contexts. Moreover, tag classes may not be necessary in which case the range of tag identifiers could be extended to 0-126.

It is difficult to determine how the automated generation of tags would affect the length and complexity of an encoding. This depends upon the algorithm employed and the individual protocols involved. Aside from this, however, it is clear that the concept of a tagged type is only meaningful to the transfer syntax; it does not contribute to the semantics of the type itself. Thus, its inclusion within the ASN.1 standard violates the separation of syntax and semantics.

5.7 Length Encodings

When encoding an ASN.1 constructed type the calculation of the length values can be problematic. Because the inner lengths determine the outer lengths, it seems appropriate to encode the inner values first. Yet, this is difficult to do since it is not clear how to determine where to begin placing these inner encodings within the encoding buffer and the exact size of this buffer will not be known until the encoding process is complete. As such, the calculation of the length values presents itself as a "chicken and egg" problem. There are three alternative methods of providing a solution.

The first alternative is to build the encodings in reverse. Specifically, the encoding process starts at the inner-most definition of the last element and proceeds outward. This requires a buffer management capability that allows buffers to grow dynamically. Unfortunately, this method precludes the possibility of directly accessing the octets of an encoding from a file. Therefore, if a large PDU were stored within a file, the entire value would have to be read into memory before the encoding process could commence.

The second alternative is to pre-compute all of the length values before encoding. The primary advantage of this method is that the exact size of the encoding is known in advance so buffer allocation is not a problem. A modest disadvantage is that this process of summing the lengths can be slow.
The third alternative is to use indefinite length encodings for all constructed types. This allows the encoding to proceed without having to sum length values, which provides an increase in efficiency. Although the total size of the encoding will not be known in advance, buffers may be chained together to handle overflow. There is an intuitive tendency to view this approach as an avoidance of the problem, shifting the burden onto the decoding process. However, we found this not to be true.

When decoding it is necessary to maintain a stack of the length values. Every time a new length value is encountered, a new value is pushed onto the stack. As each octet is processed, every element within the stack is decremented unless its value is -1. Then, as the decoding of each TLV is completed, the length on the top of the stack is checked to ensure that it is either zero or an indefinite length. If this check succeeds, the value is popped. If it fails, either a length underflow or a length overflow has occurred. With this algorithm it is clear that the use of indefinite lengths does not add any significant complexity to the decoding logic.

6. Conclusions

Our experience of implementing an ASN.1 compiler proved to be challenging. There are many aspects of ISO 8824 and ISO 8825 that require further study. Work is currently in progress within the appropriate standards organizations to improve the machine processability of both ASN.1 and BER. Thus, it is clear that the problem of data heterogeneity within OSI will remain a fertile area for research.

Acknowledgements

The Computer Networks Laboratory gratefully acknowledges the financial support provided by the Flight Data Systems Branch of NASA - Johnson Space Center, and especially the help provided by Mr. William R. Reed.

References


3 Indefinite lengths are represented as -2; this allows for the distinction between the length underflow condition and the indefinite length form.