Using ASN1C to Encode/Decode Multi-Layered Protocol Messages

Objective Systems, Inc., June 2000

Abstract

The following paper presents an example of using the Objective Systems ASN1C compiler to encode/decode a multi-layered protocol message. This example uses syntax from the 1990 X.208 ASN.1 standard. This standard was replaced by the X.680 through X.683 standards in 1994. However, the older ANY and MACRO notations presented here are still in wide use and ASN1C still supports their use.

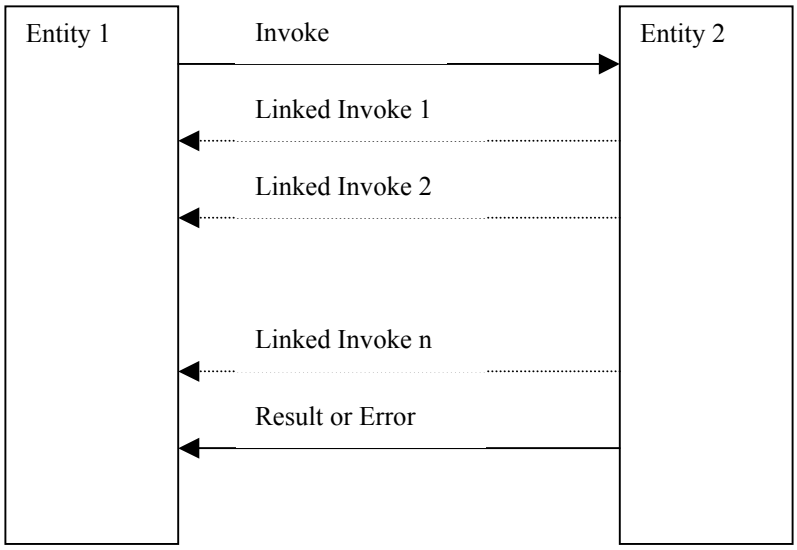
Example of a Multi-Layered Protocol

A multi-layered protocol specifies that the encoding or decoding of ASN.1 messages occur in stages or within different layers of a protocol stack. The most common analogy in traditional message processing is that of a message containing a header and a body. In one processing stage the body is composed. This then is passed to a lower layer where a standard header is applied and the message transmitted.

On the receiving end, the initial layer that processes the message would first parse the header and process the fields in which it was interested. It would then pass the body information up to a higher-level application layer that could deal with the specific type of the message.

In the 1990 ASN.1 standard (X.208), the mechanism for specifying generic data that could be passed to another layer was the ANY or ANY DEFINED BY keywords. There also existed something known as a "MACRO" which could be defined to specify the order of operations and how to apply the standard headers.

An example of this would be the Remote Operation Support Element (ROSE) protocol. This specified a two-way dialog for communications between cooperating processes. This dialog specified the following flow of communications:



In this diagram, Entity 1 "invokes" Entity 2 by sending an Invoke message. This Invoke message contains a set of standard fields (the header) and an application-specific body. This is a fully encoded ASN.1 message component. Entity 2 responds with a set of one or more optional Linked Invoke messages (which themselves can be encapsulations of the entire described protocol) followed by a final Result or Error message (these too can be optional but are usually included).

In terms of 1990 ASN.1, the Invoke message component might look like this:

```
Invoke ::= [APPLICATION 1] SEQUENCE {
      invokeId    INTEGER,
      operation   OPERATION,
      parameter   ANY DEFINED BY operation
}
```

In this definition, the first set of fields makes up the common header. This is added on top of all messages before they are sent to the other entity. The 'parameter' component is where the application-specific body of the message is stored.

When the ASN1C compiler comes across a definition like this, it generates an 'ASN1OpenType' mapping for the parameter field. This type contains a number of octets and data pointer which are designed to be used to hold information on a previously encoded message component. Therefore, to encode an Invoke message, the steps would be as follows:

1. Encode a message of the specific application type to be sent. This will yield an octet count and pointer to an encoded component.

2. Populate the fields of the ROSE Invoke header and encode. In this case, the header fields would first be set to the desired values. The parameter field would then be set to point at the message component encoded in step 1 (i.e. the numocts field would be set to the octet count and the data pointer would be the pointer to the encoded component).

On the decode side, the operation would be reversed:

1. Decode the ROSE header which will yield a structure containing the decoded header fields and a pointer and length to the encapsulated message component.

2. Based on the header fields, the appropriate application decoder function can be called. The octet count and data pointer from the open type field would be passed to this function.

The linked invoke, result, and error fields would be handled in a similar fashion.

The Use of Macros to Define Message Sequences

A MACRO is used to define all of the allowed Invokes, Linked Invokes, Results, and Errors that could occur in any given transaction between the cooperating entities. First a macro definition is developed to specify a syntax for message exchange definitions. The definition for our ROSE OPERATION might look like the following:

```
OPERATION MACRO ::=
BEGIN
  TYPE NOTATION           ::= Parameter Result Errors LinkedOperations
  VALUE NOTATION          ::= value (VALUE CHOICE {
                                      localValue INTEGER,
                                      globalValue OBJECT IDENTIFIER})
  Parameter               ::= ArgKeyword NamedType | empty
  ArgKeyword              ::= "ARGUMENT" | "PARAMETER"
  Result                  ::= "RESULT" ResultType | empty
  Errors                  ::= "ERRORS" "{"ErrorNames"}" | empty
  LinkedOperations        ::= "LINKED" "{"LinkedOperationNames"}" | empty
  ResultType              ::= NamedType | empty
  ErrorNames              ::= ErrorList | empty
  ErrorList               ::= Error | ErrorList "," Error
  Error                   ::= value(ERROR)        -- shall reference an error value
                            | type                -- shall reference an error type
                                                  -- if no error value is specified
  LinkedOperationNames    ::= OperationList | empty
```

```
  OperationList           ::= Operation | OperationList "," Operation
  Operation               ::= value(OPERATION)    -- shall reference an operation value
                            | type                -- shall reference an operation type
                                                  -- if no operation value is specified
  NamedType               ::= identifier type | type

END
```

The types and values that make up a specific exchange can then be defined using the MACRO syntax. For example, a login operation might be defined as follows:

```
login OPERATION
ARGUMENT SEQUENCE { username IA5String, password IA5String }
RESULT   SEQUENCE { ticket OCTET STRING, welcomeMessage IA5String }
ERRORS { authenticationFailure, insufficientResources }
::= 1
```

Note in the definition of Invoke above that 'operation' is defined to be of type 'OPERATION' and 'parameter' is defined as 'ANY DEFINED BY operation'. This provides the linkage between type definitions and a specific instance of an OPERATION. To encode any of the different transaction message types (Invoke, Result, Linked Invoke, or Error), the operation field would be populated with the value constant following the '::=' in the OPERATION definition. The ANY DEFINED BY field would be populated with the pointer and length of a previously encoded message component. This component must be of the type defined for the specified transaction within the operation definition.

Example of Encoding a Login Message Invoke Using ASN1C

The ASN1C compiler does not provide direct support for handling user defined MACROS, but customized versions are available for handling some standard MACRO types. The ROSE macro defined above is one of the supported MACRO types.

The ROSE customized version of ASN1C generates two types of items in support of ROSE macros:

1. It extracts the type definitions from within the OPERATION definitions and generates equivalent C/C++ structures and encoders/decoders, and

2. It generates value constants for the value associated with the OPERATION (i.e. the value to the right of the '::=' in the definition).

The compiler does not generate any structures or code related to the OPERATION itself (for example, code to encode the body and header in a single step). The reason is because of the multi-layered nature of the protocol. It is assumed that the user of such a protocol would be most interested in doing the processing in multiple stages, hence no single function or structure is generated.

Therefore, to encode the login example the user would do the following:

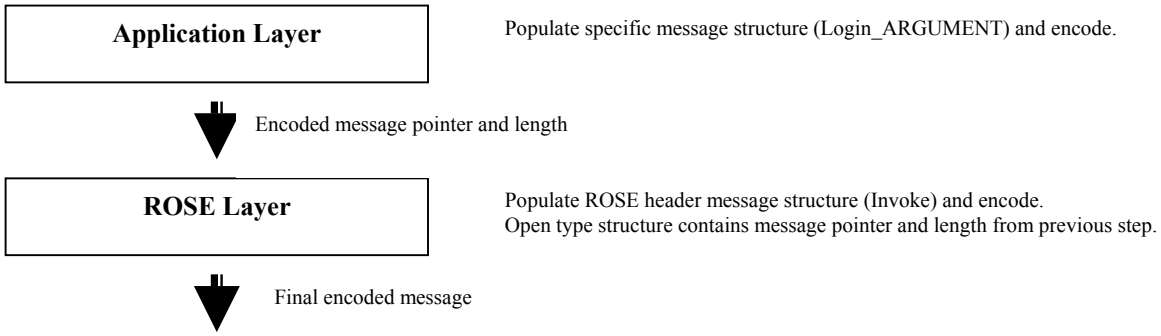
1. At the application layer, the Login_ARGUMENT structure would be populated with the username and password to be encoded.

2. The encode function for Login_ARGUMENT would be called and the resulting message pointer and length would be passed down to the next layer (the ROSE layer).

3. At the ROSE layer, the Invoke structure would be populated with the OPERATION value, invoke identifier, and other header parameters. The parameter.numocts value would be populated with the length of the message passed in from step 2. The parameter.data field would be populated with the message pointer passed in from step 2.

4. The encode function for Invoke would be called resulting in a fully encoded ROSE Invoke message ready for transfer across the communications link.

The following is a picture showing this process:



On the decode side, the process would be reversed with the message flowing up the stack:

1. At the ROSE layer, the header would be decoded producing information on the OPERATION type (based on the MACRO definition) and message type (Invoke, Result, etc..). The invoke identifier would also be available for use in session management. In our example, we would know at this point that we got a login invoke request.

2. Based on the information from step 1, the ROSE layer would know that the Open Type field contains a pointer and length to an encoded Login_ARGUMENT component. It would then route this information to the appropriate processor within the Application Layer for handling this type of message.

3. The Application Layer would call the specific decoder associated with the Login_ARGUMENT. It would then have available to it the username/password the user is logging in with. It could then do whatever application-specific processing is required with this information (database lookup, etc.).

4. Finally, the Application Layer would begin the encoding process again in order to send back a Result or Error message to the Login Request.

A picture showing this is as follows: