ASN1C Support for Information Objects and Parameterized Types

Objective Systems, Inc., April 2002

Abstract

The ASN1C ASN.1 compiler is capable of parsing advanced ASN.1 syntax as defined in the X.681 (Information Objects) and X.683 (Parameterized Types) standards. On the back-end, C, C++, or Java structures and code are generated for some of these constructs. This white paper attempts to explain the extent of the support provided by the Objective Systems ASN.1 compilers in these areas. As always, the methodology for determining this support was studying actual ASN.1 specifications to see how these items were used in practice.

Information Objects and Classes are used to define multi-layer protocols in which "holes" are defined within ASN.1 types for passing message components to different layers for processing. These items are also used to define the contents of various messages that are allowed in a particular exchange of messages. The ASN1C compiler extracts the types involved in message exchanges and generates encoders/decoders for them. The "holes" in the types are accounted for by adding open type holders to the generated structures. These open type holders consist of a byte count and pointer for storing information on an encoded message fragment for processing at the next level.

Parameterized types allow parameters of any type to be passed into generic versions of ASN.1 types. These work in much the same way as templates do in C++. Although any type of entity can be passed as a parameter, ASN1C is only concerned with those parameters that modify type definitions. The two most common modifiers that have been observed are:

1. Embedded type definitions for specific types to be included inside a more generic type, and

2. Value range constraint modifiers that allow a variable upper and/or bound values to be specified.

The compilers concentrate their support for parameterized types on these cases.

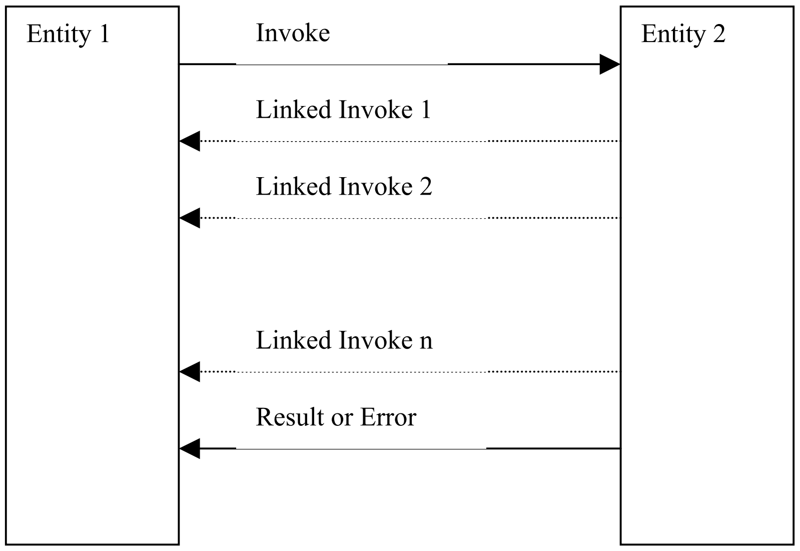
Example of a Multi-Layered Protocol

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 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. This 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 the ASN1C90 version of the compiler (included with the standard distribution) contains built-in logic for handling some standard MACRO types. The ROSE macro defined above is one of the supported MACRO types.

ASN1C90 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:

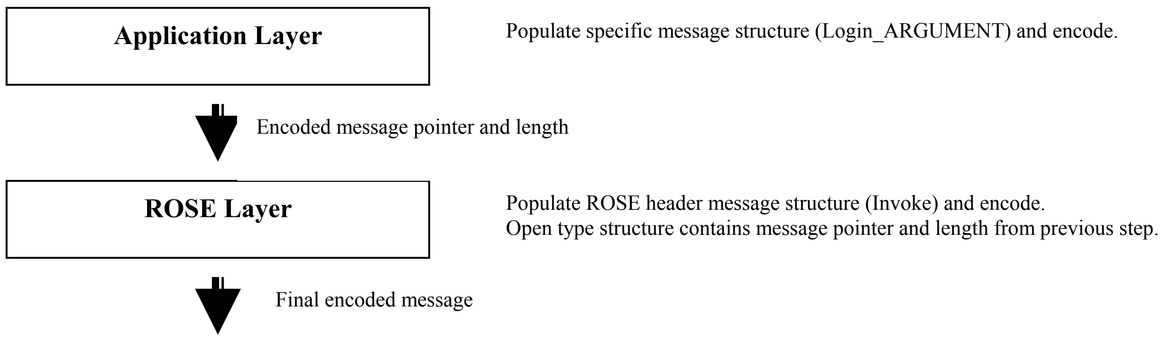
1. At the application layer, the Login_ARGUMENT structure would be populated with the username and password to be encoded.

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:


Application Layer
Populate specific message structure (Login_ARGUMENT) and encode.
Encoded message pointer and length
ROSE Layer
Populate ROSE header message structure (Invoke) and encode.
Open type structure contains message pointer and length from previous step.
Final encoded message

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

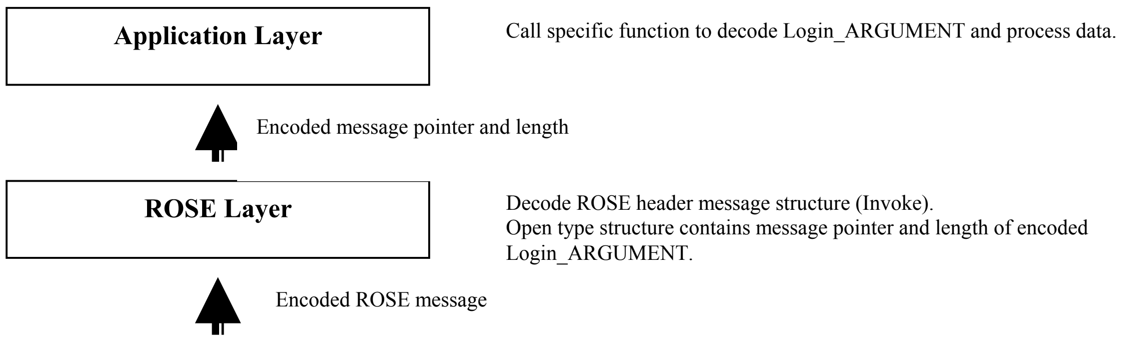
1. At the ROSE layer, the header would be decoded producing information on the OPERATION type (based on the MACRO definition) and message type (Invoke, Result, etc..). The invoke identifier would also be available for use in session management. In our example, we would know at this point that we got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


Application Layer
Call specific function to decode Login_ARGUMENT and process data.
Encoded message pointer and length
ROSE Layer
Decode ROSE header message structure (Invoke).
Open type structure contains message pointer and length of encoded Login_ARGUMENT.
Encoded ROSE message

The ROSE Protocol Defined Using Information Objects

The 1994 ASN.1 standard was the first to introduce the notion of Information Object Classes, Table Constraints, and Parameterized Types. The new standards replaced two of the fundamental elements in specifying a multi-layer protocol as defined above: the ANY type and MACRO's.

Information Object Equivalents for ANY and ANY DEFINED BY

The ANY and ANY DEFINED types were replaced with Type fields within Information Object Class definitions. In its simplest form, an ANY (without the DEFINED BY) represented an encoded component of any ASN.1 type. The equivalent element in an Information Object Class is a **Type Field**. Two distinct syntactic constructs are required to represent a simple ANY type:

1. An Information Object Class definition with a Type Field specification, and

2. A reference to this Type Field in a standard type declaration

Let's take a simplified representation of the ROSE Invoke type (a more complete representation will be presented later) and convert it to a post-1994 type using Information Object Classes. The simplified representation is as follows:

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

In this representation, operation is declared to be of type INTEGER instead of OPERATION since we are no longer using MACRO's. And the data element is a plain ANY instead of an ANY DEFINED BY operation. A simple class definition for defining the ANY type would be as follows:

```
ANY-TYPE ::= CLASS {
      &Type
}
```

This states that the type for this field must be supplied at run-time. We can then use the simple class within our Invoke type definition:

```
Invoke ::= [APPLICATION 1] SEQUENCE {
      operation   INTEGER,
      invokeId    INTEGER,
      data        ANY-TYPE.&Type
}
```

The only thing that has changed is that ANY has been replaced with the reference to ANY-TYPE.&Type. The ASN1C compiler handles both of these declarations in exactly the same way: it generates an ASN1OpenType field in the C or C++ header file for the parameter field. This is expected to contain an encoded message component from another layer as described in the Multi-Layered protocol sample above.

In practice, it is not necessary to define the ANY-TYPE class shown above in an actual ASN.1 specification. The ASN.1 X.681 standard has a built in CLASS called 'TYPE-IDENTIFIER' that can be used instead. Its definition is as follows:

```
TYPE-IDENTIFIER ::= CLASS {
      &id OBJECT IDENTIFIER UNIQUE,
      &Type
}
WITH SYNTAX { &Type IDENTIFIED BY &id }
```

Using this definition, ANY-TYPE.&Type can be replaced with TYPE-IDENTIFIER.&Type and the meaning would be the same.

The other field in this class definition, &id, is an example of a **Fixed Type Field**. The difference between this and a variable type field is that when &id is referenced, it assumes the type specified in the class. So something referenced as TYPE-IDENTIFIER.&id within a type specification is of type OBJECT-IDENTIFIER. The ASN1C compiler has the capability of referencing these types in the class definitions and generating the correct typedef in the resulting C or C++ code (or class in Java).

Using the Fixed Type Field in conjunction with the Type Field within the CLASS along with table constraints allows us to redefine our original Invoke type that used ANY DEFINED BY. In this definition the ANY type that was used was related to the operation field value. This can be rewritten using the standard TYPE-IDENTIFIER class as follows:

```
Invoke ::= [APPLICATION 1] SEQUENCE {
      operation   TYPE-IDENTIFIER.&id ({OpSet}),
      invokeId    INTEGER,
      parameter   TYPE-IDENTIFIER.&Type ({OpSet}{@operation})
}
```

In this type, operation is defined to be an identifier with a table constraint that ties it to a specific set of operations. The parameter field is then defined to be an Open Type constrained to the same set of operations and indexed by the operation value. This effectively selects a row from the table and ensures that the parameter type is that assigned in the table for the given ID code.

The ASN1C compiler does not provide support for this type of parameter matching (note: this is planned for a future release, see the *Future Plans* section later in this document). The main reason is because of the difficulty in verifying that an encoded component is of a given type. Remember that the open type field is populated with a message component that has already been encoded in another layer. So in order to verify that the parameter is of the type specified in the table, an encoder or decoder would somehow need to work backwards to determine the type from the encoding. In BER, it may be possible to do this by examining the initial tag on the component. But this assumes that the initial tag is unique which may not necessarily be the case. In PER, it is practically impossible.

There are other types of fields besides Type Fields and Fixed Type Fields that can be declared in Information Object Class definitions. The ASN1C compiler parses and silently ignores these fields because they have no effect on the generated C or C++ type definitions.

The ROSE OPERATION Macro as an Information Object Class

The preceding example defined the ROSE Invoke type in terms of the TYPE-IDENTIFIER built-in CLASS to illustrate the equivalent mechanisms for defining ANY and ANY DEFINED BY. But there is more to the ROSE protocol then a simple ID and associated type. For any given ID, there are multiple message types that are exchanged for Invoke, Linked Invoke, and Result messages. There are also ERRORS that have there own ID's and associated types. An example of CLASS definitions that might be used to model the ROSE protocol is as follows [1]:

```
OPERATION ::= CLASS {
      &operation   CHOICE { local  INTEGER,
                            global OBJECT IDENTIFIER } UNIQUE,
      &ArgumentType,
      &ResultType,
      &Errors   ERROR OPTIONAL
  }

ERROR ::= CLASS {
      &errorCode    CHOICE { local  INTEGER,
                            global OBJECT IDENTIFIER } UNIQUE,
      &ParameterType OPTIONAL
}
```

This CLASS defines an OPERATION as being composed of an operation identifier, an argument type, a result type, and errors (which are optional). If you go to the diagram at the top of this document, you can get a feeling for what these different components represent (note: Linked Invoke was not included in this example to keep it simpler). In general, an encoded message of a given type corresponding to an operation identifier is passed in the invoke message to the other entity. The other entity is then expected to reply with either the expected result message or one of the defined errors for the operation.

We will ignore errors to keep the example simpler. But they follow the same rules as the operation objects that will be presented next. The information objects for operations define the legal argument and result types that can be exchanged for a given operation. So for our login example above, the information object would look like this:

```
login OPERATION ::= {
    &operation    local : 1,
    &ArgumentType SEQUENCE { username IA5String, password IA5String },
    &ResultType SEQUENCE { ticket OCTET STRING, welcomeMessage IA5String },
    &Errors { authenticationFailure, insufficientResources }
}
```

It is then possible to declare an Information Object Set that declares all of the operations the software will support. In our simple example, we will assume we are only supporting login operation and a logoff operation (the logoff operation is not defined here but would be done the same way as login). The object set would be defined as follows:

```
My-ops OPERATION ::= { login | logoff }
```

We can now write our Invoke header type again in terms of the operation set we have defined:

```
Invoke ::= SEQUENCE {
    invokeId      INTEGER,
    operation     OPERATION.&operation ({My-ops}),
    argument      OPERATION.&ArgumentType ({My-ops}{@opcode}) OPTIONAL
}
```

This is similar to the type presented in the previous section but now has been written in terms of our complete ROSE protocol specification.

ASN1C Support for Information Objects

ASN1C can parse the information objects defined above and will extract information that can be useful in the generation of messages of these types.

First, let us examine the login information object:

```
login OPERATION ::= {
    &operation    local : 1,
    &ArgumentType SEQUENCE { username IA5String, password IA5String },
    &ResultType SEQUENCE { ticket OCTET STRING, welcomeMessage IA5String },
       &Errors { authenticationFailure, insufficientResources }
}
```

In this case, the object contains three items of interest:

1. An argument type (sequence of username and password)
2. A result type (sequence of ticket and welcomeMessage)
3. An operation code (local : 1) that will be stored in the ROSE header

The ASN1C compiler will extract these embedded items from the message and generate source code for them in the target language.

In the case of the argument and result types, it will extract the definitions and form ASN.1 productions of the following general form:

```
<infoObjectName>_<fieldname> ::= <type>
```

The '&' is removed from the field name to form a valid target language definition.

So, for example, the &ArgumentType and &ResultType fields above will be converted to the following equivalent ASN.1 productions:

```
login_ArgumentType ::= SEQUENCE { username IA5String, password IA5String }

login_ResultType ::= SEQUENCE { ticket OCTET STRING, welcomeMessage IA5String }
```

(These ASN.1 definitions are not technically correct because they begin with a lowercase letter and contain an underscore – but the compiler allows these definitions in this specific case). The resulting typedef for login_ArgumentType for C would be as follows:

```
typedef struct login_ArgumentType {
   ASN1IA5String username;
   ASN1IA5String password;
} login_ArgumentType;
```

A similar typedef would be generated for login_ResultType and encode and decode functions would be generated for both types. The results would be similar if the target language was C++ or Java.

In the case of an embedded value definition, the value is extracted and a value production of the following general form is created:

```
<infoObjectName>_<fieldname> <valueType> ::= <value>
```

In the case of 'operation', an ASN.1 value definition of the following form would be generated:

```
login_operation CHOICE { local  INTEGER, global OBJECT IDENTIFIER } ::= 1
```

In this case, the compiler had to go back to the original class definition to find the type of the operation value (i.e <valueType>) in the definitions above. In the case of C, this would lead to the following #define constant being generated:

```
#define ASN1V_operation 1
```

ASN1C Support for Information Object References in Types

We now turn our attention to the Invoke type that references fields within the Information Object class:

```
Invoke ::= SEQUENCE {
       invokeId      INTEGER,
       operation     OPERATION.&operation ({My-ops}),
       argument      OPERATION.&ArgumentType ({My-ops}{@opcode}) OPTIONAL
}
```

In this case, in order to generate the proper type definition for this construct, the compiler must have the capability to refer back to the class definition to include the proper type references in the generated code. The ASN1C compiler is capable of doing this and will generate the following C typedef (or equivalent Java class) for the type above:

```
typedef struct Invoke {
   ASN1INT invokeId;
   OPERATION_operation operation;
   ASN1OpenType argument;
} Invoke;
```

The shortcoming of the compiler in this case is that it does not enforce the table constraint imposed on the operation and argument fields. The user is trusted to populate the generated structure with one of the operation values defined within the information object set and then to supply an encoded value of the type corresponding to that operation. Possible improvements in this area are discussed in the *Future Plans* section later in the document.

The Use of Parameterization with the ROSE Information Object

The example in the last section showed a complete information object specification for a simplified ROSE protocol. But one piece is still missing. It would be nice if the type specifications for Invoke and the other header types used in the protocol were common instead of tied to a specific set of Information Objects as they are above. Recall that the Invoke type contains a reference to "My-ops" in the table constraint added to the object. It would be expected that other users of the ROSE protocol would have different sets of information objects that they would want used in this way. Some way is needed to pass the operation set into a common representation of the type.

The mechanism to achieve this is known as parameterization. It is basically a substitution mechanism that allows a dummy parameter specified on the left-hand side of the type definition to be passed in each time the type is used. In the case of the Invoke type, we would want to be able to pass in an information object set. We would do this by defining the type as follows:

```
       Invoke {OPERATION: User-ops} ::= {
              invokeId  INTEGER,
              operation OPERATION.&operation ({User-ops}),
              argument  OPERATION.&ArgumentType ({User-ops}{@opcode}) OPTIONAL
       }
```

And then a reference to the type would be as follows:

```
       My-invoke ::= Invoke {My-ops}
```

The ASN1C compiler in its current state would not be concerned with an information object set passed in this fashion. The reason is because it does nothing with the object set anyway, so the parameter is discarded to allow a simple type to be formed. An option to provide more advanced support is being considered (see the *Future Plans* section later in the document).

Parameterization can be used for a lot more then just information object sets as discussed above. Just about anything can be passed as a parameter and this allows all sort of common definitions to be created. In general, ASN1C can recognize when parameterization is being used, but it will only affect the generated code when it causes a type definition to be modified. Two common uses that ASN1C supports are:

1. The inclusion of one type within another (an example of this is the SIGNED parameterized type used in security specifications), and

2. The setting of constraint bounds (most commonly the upper bound) through a parameter. An example of this would be SizedString { INTEGER : ub } ::= IA5String SIZE (1..ub).

An Example From 3GPP

The 3GPP ASN.1 specifications make use of the Information Object and Parameterization concepts presented above. This section will show snippets of code from one of these specifications and discuss what ASN1C does and doesn't do. The snippets are from the RANAP specification and can be found in the infoObject sample program that comes with the compiler demo version.

First, a class is defined which will be the basis for all other definitions:

```
RANAP-PROTOCOL-IES ::= CLASS {
       &id                        ProtocolIE-ID       UNIQUE,
       &criticality               Criticality,
       &Value,
       &presence                  Presence
}
WITH SYNTAX {
       ID                         &id
       CRITICALITY                &criticality
       TYPE                       &Value
       PRESENCE                   &presence
}
```

This class is very similar to the TYPE-IDENTIFIER class. The only difference is that two additional elements are defined – criticality and presence. The id, criticality, and presence fields are all fixed type fields. They assume the types of ProtocolIE-ID, Criticality, and Presence that are defined as follows:

```
ProtocolIE-ID   ::= INTEGER (0..65535)

Criticality   ::= ENUMERATED { reject, ignore, notify }

Presence      ::= ENUMERATED { optional, conditional, mandatory }
```

So whenever the compiler sees a reference to any of these fields within a type specification, it resolves the type to the corresponding fixed type from within the class.

The &Value field is a type field. This is translated to an open type by the compiler when it is referenced in a type specification.

As mentioned previously, ASN1C does not generate any code for a CLASS specification. But it will parse and internally store the definition so that it can refer back to it to resolve later references to items within it.

The next item of interest is the ProtocolIE-Field type definition:

```
ProtocolIE-Field {RANAP-PROTOCOL-IES : IEsSetParam} ::= SEQUENCE {
   id                RANAP-PROTOCOL-IES.&id            ({IEsSetParam}),
   criticality       RANAP-PROTOCOL-IES.&criticality   ({IEsSetParam}{@id}),
   value             RANAP-PROTOCOL-IES.&Value         ({IEsSetParam}{@id})
}
```

This is a parameterized type definition that references fields from within the class. In this case, the parameter is an information object set as it was in the case of the ROSE Invoke type presented earlier. This definition, like the class before it, is parsed but does not result in the generation of any code. It is parsed and internally stored by the compiler until a concrete instance of a type is created from it.

The next item is also a parameterized type:

```
ProtocolIE-Container {RANAP-PROTOCOL-IES : IEsSetParam} ::=
   SEQUENCE (SIZE (0..maxProtocolIEs)) OF ProtocolIE-Field {{IEsSetParam}}
```

This is similar to the type definition above. It is simply a container type definition to allow multiple protocol-ID fields to be passed in a message definition. As with the other parameterized type, the compiler parses and remembers its structure.

An information object set is defined to tell us what we are allowed to pass in a transaction:

```
Iu-ReleaseCommandIEs RANAP-PROTOCOL-IES ::= {{
   ID           id-Cause
   CRITICALITY  ignore
   TYPE         Cause
   PRESENCE     mandatory
},
...
}
```

The compiler parses but does nothing further with this specification. The user must use this to assemble or disassemble the message components. This defines the multi-layered protocol. It says that for ID = id-Cause, the Cause type is to be used. Cause is a normal ASN.1 type specified at some other point in the specification.

Finally, we get to a concrete type definition:

```
Iu-ReleaseCommand ::= SEQUENCE {
   protocolIEs      ProtocolIE-Container { {Iu-ReleaseCommandIEs} },
   ...
}
```

All the parts are referenced here. The ProtocolIE-Container is the parameterized type described above and we are passing the Iu-ReleaseCommandIEs information object set. The ASN1C compiler has kept track of the entire chain from the class definition through the two parameterized types to come up with the correct type definitions for this message:

```
typedef struct EXTERN ProtocolIE_Field {
   ProtocolIE_ID  id;
   Criticality  criticality;
   ASN1OpenType  value;
} ProtocolIE_Field;

typedef struct EXTERN ProtocolIE_Container {
   ASN1UINT n;
   ProtocolIE_Field *elem;
} ProtocolIE_Container;
```

In this case, the types generated are taken directly from the parameterized type definitions. Since the compiler has determined that the information object set parameter is not to be used for anything, it is simply discarded and the parameterized types are turned into simple types.

So how would a message of this type be encoded? The reader would first have to go to the information object set specification to see what must be encoded at each layer. Recall that this specification is as follows:

```
Iu-ReleaseCommandIEs RANAP-PROTOCOL-IES ::= {{
   ID           id-Cause
   CRITICALITY  ignore
   TYPE         Cause
   PRESENCE     mandatory
},
...
}
```

This says that the open type field (TYPE in the definition) for this instance is of type 'Cause'. So at the first layer a message of type Cause is encoded. This is the body of the message. Then at the next layer, the header is applied. This is done by plugging the encoded message pointer and length into the 'value' field in the Iu_ReleaseCommand_protocolIEs_aSeq type. The id field would then be set to id_Cause and the criticality field would be set to ignore. The remaining fields would then be set to encode one element in the SEQUENCE OF definition. Finally, the generated encode function would be called to encode the final message to be transmitted.

The decode process would work in the opposite way. The specific header decode function would first be called to decode the header. From there, the application can get at the id field to determine what type of transaction was received. It could then pass the embedded message pointer and length down to the appropriate function at the next layer to handle the type of message received.

The complete example of encoding and decoding an id-Cause transaction can be found in the writer and reader programs in the infoObject sample directory that comes with the compiler.

Future Plans

As we have seen, the ASN1C compiler can parse Information Object Class definitions, Information Objects, Information Object Sets, and Parameterized Types. It can extract embedded types from these objects in order to generate code to allow variables of these types to be encoded or decoded. It can also properly create target language type and value definitions for ASN.1 types that reference fields within these objects.

The missing piece at this time is the handling of table constraints. Some applications are based on communication-stack-like functionality and delegate the handling of different components of the message to different layers in the stack. These would not benefit much from adding functionality to do "one-step" encoding or decoding of the entire message structure. But other applications that do everything in a single-layer can benefit from this flattening of the stack.

The basic idea in achieving one-step encoding or decoding is to maintain a second type structure with each type field that would hold the decoded value of the type (recall that the open type holder that is currently supplied holds the encoded representation). Multiple encode and decode functions or methods would then be generated to encode or decode a portion of the message. Each of these component encode or decode functions would have a numeric identifier included to indicate that it was only encoding or decoding part of the message. The overall encode or decode function or method will contain no prefix number and will consist of a series of calls to the component functions to encode or decode the entire message.

It is important to note that the generation of all of this extra logic has the potential to result in a huge increase in the amount of generated code. ASN.1 specifications such as the 3GPP RANAP and NBAP specifications contain large numbers of information objects and parameterized container definitions. The current ASN1C methodology of discarding parameters and ignoring information object sets results in lean code that can be used to accomplish the encoding or decoding of these message types. A moderate amount of coding is required by the user to use these structures. The price of reducing the user's coding burden and enforcing the table constraints will be code-bloat which may not be suitable for some environments (for example, embedded). It shall therefore remain a compiler option to do the type of code generation detailed below.

To illustrate what is to be done, we can use the 3GPP RANAP message defined above as an example. We will walkthrough what will be done for C to provide support for the table constraint variables. We begin by looking at the ProtocolIE-Field type defined above.

```
ProtocolIE-Field {RANAP-PROTOCOL-IES : IEsSetParam} ::= SEQUENCE {
   id               RANAP-PROTOCOL-IES.&id           ({IEsSetParam}),
   criticality      RANAP-PROTOCOL-IES.&criticality  ({IEsSetParam}{@id}),
   value            RANAP-PROTOCOL-IES.&Value        ({IEsSetParam}{@id})
}
```

The generated typedef that currently exists for this type is as follows:

```
typedef struct EXTERN ProtocolIE_Field {
   ProtocolIE_ID  id;
   Criticality  criticality;
   ASN1OpenType  value;
} ProtocolIE_Field;
```

In this case, the IEsSetParam parameter was simply discarded because the compiler had no use for it. This yields a simple type (ProtocolIE-Field) that can now be referenced in any other place it is used.

To make the proposed changes easier to understand, we provide a simplified version of the Iu-ReleaseCmd that was presented earlier:

```
Iu-ReleaseCommand ::= SEQUENCE {
   protocolIEs      ProtocolIE-Field { {Iu-ReleaseCommandIEs} },
   ...
}
```

In this case, we directly reference the ProtocolIE-Field parameterized type allowing only a single field in our message (recall before we referenced a ProtocolIE-Container type that allowed a sequence of these field types).

The first item in the proposed modification is the use of the information object set parameter. This changes what was a simple reference to a ProtocolIE-Field type above to the creation of a new type instance that takes the parameter into account. Using the normal rules for the creation of a temporary type, this new type definition would be as follows:

```
typedef struct EXTERN Iu_ReleaseCommand_protocolIEs {
   ProtocolIE_ID  id;
   Criticality  criticality;
   ASN1OpenType  value;
   void* decoded_value;
} Iu_ReleaseCommand_protocolIEs;
```

The two things to notice are:

1. A new type was created (Iu_ReleaseCommand_protocolIEs) instead of a simple reference to the ProtocolIE_Field type, and

2. A new field has been added – decoded_value – that will hold a pointer to a type variable of the ASN.1 type that corresponds to the id field for any given object in the information object set.

Note that the type of decoded_value is a void pointer. This is because this field could hold a pointer to a potentially large number of types. It would have been possible to set up a union structure to all of the different possible pointer types using information from the information object set (as we do for a CHOICE construct) but it was felt that this would add excessive bulk and provide little additional information. It will be the user's responsibility to ensure that the type of value populated in this field is consistent with the id field for encoding or to properly cast the pointer to operate on the data after decoding.

The next change will involve how encode and decode functions are generated. Instead of a single encode and decode function, there will be three of each:

1. An encode and decode function (method) to encode/decode the header. This will have a postfix of 1 applied to the normal prefix (for example asn1PE1_<type>). This will be identical to the function (method) that was previously generated.

2. An encode and decode function (method) to encode/decode the body of the message. This will have a postfix 2 applied to the normal prefix (for example, asn1PE2_<type>). This function will select and execute the appropriate encode/decode function for the type of the payload being processed based on the value in the id field.

3. A "one-step" encode and decode function that will invoke the previous two functions in turn to accomplish a complete encode or decode of the message.

Pseudo-code for the encode functions is as follows (note: item 2 is presented first because the normal encoding procedure would be to first encode the body followed by the header):

```
asn1PE2_<type> (ASN1CTXT* ctxt_p, <type>* pvalue, …)
{
   if (id == <objsetID1>) {
      asn1PE_<objsetType1> (ctxt_p, (<objsetType1>*)pvalue->decoded_value, …);
   }
   else if (id == <objsetID2>) {
      asn1PE_<objsetType2> (ctxt_p, (<objsetType2>*)pvalue->decoded_value, …);
   }
   ...
   pvalue->value.numocts = encoded length;
   pvalue->value.data = encode buffer pointer;
}
```

The user is required to have populated a variable of the body type with the data to be encoded and then set the decoded_value pointer within the structure to point to this item. It is also required that the user populates the header id field with the identifier for this item. The encode function will then invoke the proper encode function based on the identifier value. The resulting encode buffer data parameters are then stored in the open type structure for the value.

The second function would be identical to the function currently generated. It would encode the header fields and encoded open type value:

```
asn1PE1_<type> (ASN1CTXT* ctxt_p, <type>* pvalue, …)
{
   encode header fields
   encode open type value
}
```

The main function will then call both of these functions to accomplish the complete encoding of a message in one step:

```
asn1PE_<type> (ASN1CTXT* ctxt_p, <type>* pvalue, …)
{
   save current encoding context
   set encode buffer to point to temp buffer
   call asn1PE2_<type>
   restore original encode context
   call asn1PE1_<type>
}
```

Decoding would be accomplished by simply reversing the process. Function 1 would first be invoked to decode the header and set the open type variable to hold the encoded data parameters. Function 2 would then be invoked to decode the body. This is accomplished by using the decoded identifier value to select the proper decode function to invoke. Memory to hold the decoded value would be allocated within the function. The main function would then consist of calls to each of these functions in turn to accomplish the complete decoding.

Summary

This paper has attempted to present what the ASN1C compiler does to handle Information Objects and Parameterized Types when encountered in an ASN.1 specification. They key thing to keep in mind is that the compiler is trying to get at the underlying types that form the components of the messages to be exchanged. To this end, it is capable of parsing Class and Parameterized Type definitions and keeping track of the information required to generate accurate types and encoders/decoders when concrete type definitions reference elements from within these structures.

Since this paper was originally published in the year 2000, support has been added to the compiler to parse information object specifications (including those formed using a customized syntax specified with the WITH SYNTAX clause). Embedded types and values are now extracted and used to generate type and value definitions in the target language. This allows components defined within the objects to be encoded and decoded.

The only thing the ASN1C compiler currently does not do is the generation of code to use table constraints to accomplish “one step” encoding or decoding. Our future plans are to support this feature. The support will be governed by a command line switch or configuration parameter to allow it to be turned off if not desired. It is anticipated support of table constraints and one-step encode/decode will cause a considerable extra amount of source code to be generated, so it may not be beneficial in all situations.

References

(1) *ASN.1 Complete*, John Larmouth, Morgan Kaufmann Publishers, 2000