ASN1C C/C++
Code Generation for
3GPP and LTE Specifications

Objective Systems, Inc., December 2008

3GPP ASN.1 Code Generation Design

Many of the ASN.1 specifications used in the 3G phone system such as *Radio Access Network Application Part* (RANAP), *Node B Application Protocol* (NBAP), etc. have a common design pattern. This is also true of specifications for the new 4G (LTE) systems such as those used in the S1 and X2 protocol specifications.

The way ASN1C currently generates code for these specifications is workable, but difficult to use because it fails to take advantage of the common patterns in the syntax to provide a reduced representation. Instead, generic table constraint processing code is generated which is cumbersome to use. The goal of this project is provide a reduced code representation that is unique to 3GPP specifications. This will be specified by the user on the command-line with a new command-line switch: **-3gpp**. If this switch is set, the procedures described within this document will be performed.

Limitations

3GPP specifications generally only use the Packed Encoding Rules (PER), therefore, these will be the only rules initially supported in these modifications. XML code generation may be supported in the future.

C Example

The following is a walk through of a single message within the LTE S1AP ASN.1 specification. It shows what code is generated now (in C) and what changes are desired.

S1AP-PDU Type

At the top level is the main PDU type:

```
S1AP-PDU ::= CHOICE {
      initiatingMessage InitiatingMessage,
      successfulOutcome SuccessfulOutcome,
      unsuccessfulOutcome     UnsuccessfulOutcome,
      ...
}
```

From this, we generate the following C structure:

```
typedef struct EXTERN S1AP_PDU {
   int t;
   union {
      /* t = 1 */
      InitiatingMessage *initiatingMessage;
      /* t = 2 */
      SuccessfulOutcome *successfulOutcome;
```

```
      /* t = 3 */
      UnsuccessfulOutcome *unsuccessfulOutcome;
      /* t = 4 */
      ASN1OpenType *extElem1;
   } u;
} S1AP_PDU;
```

This is the standard structure for a CHOICE and is not expected to change.

InitiatingMessage Type

If we then look at the InitiatingMessage option, we see the following structure:

```
InitiatingMessage ::= SEQUENCE {
      procedureCode     S1AP-ELEMENTARY-PROCEDURE.&procedureCode
                              ({S1AP-ELEMENTARY-PROCEDURES}),
      criticality S1AP-ELEMENTARY-PROCEDURE.&criticality
                              ({S1AP-ELEMENTARY-PROCEDURES}{@procedureCode}),
      value       S1AP-ELEMENTARY-PROCEDURE.&InitiatingMessage
                              ({S1AP-ELEMENTARY-PROCEDURES}{@procedureCode})
}
```

Types and codes for this type are defined in the S1AP-ELEMENTARY-PROCEDURE Information Object Set. There are CLASS-1 and CLASS-2 procedures defined. The full list specification is as follows:

```
S1AP-ELEMENTARY-PROCEDURES S1AP-ELEMENTARY-PROCEDURE ::= {
      S1AP-ELEMENTARY-PROCEDURES-CLASS-1       |
      S1AP-ELEMENTARY-PROCEDURES-CLASS-2,
      ...
}

S1AP-ELEMENTARY-PROCEDURES-CLASS-1 S1AP-ELEMENTARY-PROCEDURE ::= {
      handoverPreparation                |
      handoverResourceAllocation         |
      pathSwitchRequest                  |
      sAEbearerSetup                     |
      sAEbearerModify                    |
      sAEbearerRelease                   |
      initialContextSetup                |
      handoverCancel                     |
```

```
      reset                                |
      s1Setup                              |
      uEContextModification                |
      uEContextRelease                     |
      eNBConfigurationUpdate               |
      mMEConfigurationUpdate               ,
      ...
}

S1AP-ELEMENTARY-PROCEDURES-CLASS-2 S1AP-ELEMENTARY-PROCEDURE ::= {
      handoverNotification                 |
      sAEbearerReleaseRequest              |
      paging                               |
      downlinkNASTransport                 |
      initialUEMessage                     |
      uplinkNASTransport                   |
      errorIndication                      |
      nASNonDeliveryIndication             |
      uEContextReleaseRequest              |
      downlinkS1cdma2000tunneling          |
      uplinkS1cdma2000tunneling            |
      uECapabilityInfoIndication           |
      eNBStatusTransfer                    |
      mMEStatusTransfer                    |
      deactivateTrace                      |
      traceStart                           |
      traceFailureIndication               |
      locationReportingControl             |
      locationReportingFailureIndication   |
      locationReport                       ,
      ...
}
```

Code as is currently generated for the InitiatingMessage type does not take the information object definitions into account up front. Types are just extracted for the fixed type fields (id and criticality) and an open type is used for the value field:

```
typedef struct EXTERN InitiatingMessage {
   ProcedureCode procedureCode;
   Criticality criticality;
```

```
   Asn1Object value;
} InitiatingMessage;
```

The use of an open type (Asn1Object) for the value field makes using the generated code difficult for users. It requires going back to the specification to figure out the type of data to be used in the field and also what to populate the *procedureCode* and *criticality* fields with.

This code generation has been changed to create a union of all of the possible value types for the value field. This union will contain comments indicating the *procedureCode* and *criticality* for each of the alternative types. In other words, the end result will be similar to having declared the value field to be a CHOICE of all of the information object set items.

An example of what the modified type would look like is as follows:

```
typedef enum {
   T1_handoverPreparation,
   T1_handoverResourceAllocation,
   T1_pathSwitchRequest,
   etc..
} S1AP_ELEMENTARY_PROCEDURE_TVALUE;

typedef struct EXTERN InitiatingMessage {
   ProcedureCode procedureCode;
   Criticality criticality;

   /**
    * information object selector
    */
   S1AP_ELEMENTARY_PROCEDURE_TVALUE t;

   /**
    * S1AP-ELEMENTARY-PROCEDURE information objects
    */
   union {
      /**
       * procedureCode: id-HandoverPreparation
       * criticality: reject
       */
      HandoverRequired*  handoverPreparation;
```

```
      /**
       * procedureCode: id-HandoverResourceAllocation
       * criticality: reject
       */
      HandoverRequest*  handoverResourceAllocation;

      /**
       * procedureCode: id-HandoverNotification
       * criticality: ignore
       */
      HandoverNotify*  handoverNotification;

      etc..

  } u;
} InitiatingMessage;
```

As can be seen, all of the possible typed values are now enumerated and comments are added to show the fixed-value fields for each alternative.

handoverPreparation InitiatingMessage

We now proceed to look in detail at the specific *InitiatingMessage* associated with the *handoverPreparation* information object. In particular, we want to look at the value field since the type of this field is determined by the *handoverPreparation* object definition. This definition is as follows:

```
handoverPreparation S1AP-ELEMENTARY-PROCEDURE ::= {
      INITIATING MESSAGE      HandoverRequired
      SUCCESSFUL OUTCOME      HandoverCommand
      UNSUCCESSFUL OUTCOME    HandoverPreparationFailure
      PROCEDURE CODE          id-HandoverPreparation
      CRITICALITY             reject
}
```

To decipher this, one would need to look at the S1AP-ELEMENTARY-PROCEDURE class definition and accompanying WITH SYNTAX specification to determine the syntax. However, in this case, the syntax is rather self explanatory. Basically it indicates the type for the *InitiatingMessage* value field is *HandoverRequired*. It also provides information on the types for response message fields and the *procedureCode* and *criticality* values. Looking at the *HandoverRequired* type definition, we see the following:

```
HandoverRequired ::= SEQUENCE {
      protocolIEs       ProtocolIE-Container      { { HandoverRequiredIEs} },
      ...
}
```

This is a parameterized type which passes an information object set (*HandoverRequiredIEs*) as a parameter. It is defined as follows:

```
ProtocolIE-Container {S1AP-PROTOCOL-IES : IEsSetParam} ::=
      SEQUENCE (SIZE (0..maxProtocolIEs)) OF ProtocolIE-Field {{IEsSetParam}}

ProtocolIE-Field {S1AP-PROTOCOL-IES : IEsSetParam} ::= SEQUENCE {
      id          S1AP-PROTOCOL-IES.&id              ({IEsSetParam}),
      criticality S1AP-PROTOCOL-IES.&criticality     ({IEsSetParam}{@id}),
      value       S1AP-PROTOCOL-IES.&Value           ({IEsSetParam}{@id})
}
```

Currently, a generic list of structures is generated for each of the types that have *protocolIEs* as an element. For example, for *HandoverRequired* we have:

```
typedef struct EXTERN HandoverRequired {
   HandoverRequired_protocolIEs protocolIEs;
   OSRTDList extElem1;
} HandoverRequired;

/* List of HandoverRequired_protocolIEs_element */
typedef OSRTDList HandoverRequired_protocolIEs;

typedef struct EXTERN HandoverRequired_protocolIEs_element {
   ProtocolIE_ID id;
   Criticality criticality;
   Asn1Object value;
} HandoverRequired_protocolIEs_element;
```

The *HandoverRequired_protocolIEs_element* contains an open type field (value) that would require the developer to find all the possible options. This can be replaced with a union structure as was done in the *InitiatingMessage* structure above. This would result in the following type definition:

```
typedef struct EXTERN HandoverRequired_protocolIEs_element {
```

```
   ProtocolIE_ID id;
   Criticality criticality;
   struct {
      /**
       * information object selector
       */
      HandoverRequiredIEs_TVALUE t;

      /**
       * HandoverRequiredIEs information objects
       */
      union {
         /**
          * id: id-MME-UE-S1AP-ID
          * criticality: reject
          * presence: mandatory
          */
         MME_UE_S1AP_ID  *_HandoverRequiredIEs_1;
         /**
          * id: id-HandoverType
          * criticality: reject
          * presence: mandatory
          */
         HandoverType  *_HandoverRequiredIEs_2;
         /**
          * id: id-Cause
          * criticality: ignore
          * presence: mandatory
          */
          ...
      } u;
   } value;
} HandoverRequired_protocolIEs_element;
```

The user would then need to allocate objects of this structure, populate them, and add them to the protocol IE list. It would be useful if helper functions could be generated in C to assist in adding information items to the list. For this purpose, an *asn1Append* function is generated in the following format:

```
asn1Append_HandoverRequired_protocolIEs_1
```

The number at the end is a sequence number to delineate each of the different type field options in the associated information object set.

The full prototype for a generated list append function is as follows:

```
int asn1Append_<ListType>_<SeqNo>
   (OSCTXT* pctxt, <ListType>* plist, <ValueType> value);
```

In this definition, <ListType> would be the type of the list (*HandoverRequired_protocolIEs* in the example above), <SeqNo> would be the sequence number of the corresponding item in the information object set, and <ValueType> would be the type as defined in the type field for the value. If the value is of a structured type, a pointer to a value would be passed rather than the value itself. The argument name in this case would be *pvalue*.

In addition to the list append function, a list get function is generated to help a reader get a specific item from the list using the key value identified in the table constraint. The format of the prototype for this function is as follows:

```
<ListElemType>* asn1Get_<ListType> (<KeyFieldType> <key>, <ListType>* plist);
```

In this definition, the following bracketed (<>) items would be replaced:

<ListItemType>	The type of an element in the IE list.
<ListType>	The type of the list.
<KeyFieldType>	The type of the key field as defined in the table constraint.
<key>	The name of the key element.

For example, for the *Handover_protocolIEs* list defined above, the following asn1Get function would be generated:

```
HandoverRequired_protocolIEs_element* asn1Get_HandoverRequired_protocolIEs
   (ProtocolIE_ID id, HandoverRequired_protocolIEs* plist)
```

Note that the function above assumes that there is only a single unique item in the list with the given ID and will therefore always fetch the first item encountered. If there are multiple items with a given key value, the user will need to manually iterate through the list to find all occurrences.

C++ Example

For C++, some optimizations of the structures generated for C are possible. This section discusses some of the items that are changed.

Choice Selector TVALUE Type

For C, an enumerated type is generated for each of the options in a type field union. These correspond to each of the items in the information object set associated with the union. For example, the TVALUE

type generated for S1AP_ELEMENTARY_PROCEDURES is as follows:

```
typedef enum {
   T1_UNDEF_,
   T1_handoverPreparation,
   T1_handoverResourceAllocation,
   T1_pathSwitchRequest,
   ...
} S1AP_ELEMENTARY_PROCEDURES_TVALUE;
```

Note the number '1' following the leading T in each of these enumerations. That number is a sequential type number used to ensure that no name clashes occur between enumerations with common names.

For C++, this type is generated as a class with TVALUE as a public member inside:

```
class S1AP_ELEMENTARY_PROCEDURES {
public:
   enum TVALUE {
      T_UNDEF_,
      T_handoverPreparation,
      T_handoverResourceAllocation,
      T_pathSwitchRequest,
   ...
   } ;
} ;
```

In this case, the type number identifier is not needed because the class name provides for unambiguous enumerated item names.

Generated Helper Functions

For C, special *asn1Append_<name>* and *asn1GetIE_<name>* functions were generated to help a user append information elements (IE's) to a list and get an indexed IE respectively. For C++, these are added as methods to the generated control class for the list type.

For example, for the *HandoverRequired_protocolIEs* type, the following methods are added to the control class:

```
class EXTERN ASN1C_HandoverRequired_protocolIEs : public ASN1CSeqOfList
```

```
{
   ...
   /* Append IE 1 with value type ASN1T_MME_UE_S1AP_ID to list */
   int AppendIE1 (ASN1T_MME_UE_S1AP_ID value);

   /* Append IE 2 with value type ASN1T_HandoverType to list */
   int AppendIE2 (ASN1T_HandoverType value);
   ...
   /* Get IE using id key value */
   ASN1T_HandoverRequired_protocolIEs_element* GetIE (ASN1T_ProtocolIE_ID id);

} ;
```

Sample code that can be used to populate a list of IE's might look as follows:

```
   ASN1C_HandoverRequired_protocolIEs
      protocolIEs (handoverRequired.protocolIEs);

   protocolIEs.AppendIE1 (123456);
   protocolIEs.AppendIE2 (HandoverType::ltetogeran);
```

Note that the user does not need to populate *procedureCode* or *criticality* when adding these items. The logic to populate items with the fixed values as defined in the ASN.1 specification is already built in.