The Design of C++ Class Hierarchy to Model ASN.1 Data Types

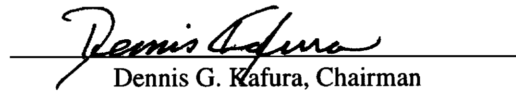
by

Rajesh Khera

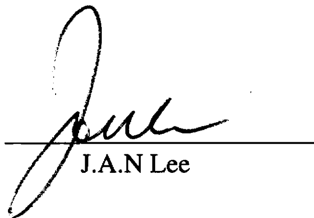
Project and Report Submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

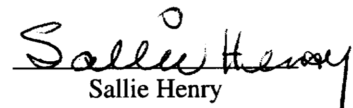
APPROVED:



Dennis G. Kafura, Chairman



J.A.N Lee



Sallie Henry

January, 1994
Blacksburg, Virginia

C.2

LD
5655
V851
1994
K447
C.2

The Design of C++ Class Hierarchy to Model ASN.1 Data Types

By : Rajesh Khera

Chairman : Dr. Dennis Kafura

Department : Computer Science

(ABSTRACT)

Computing in a heterogeneous environment requires an exchange of information among computers which have different ways of representing information. This type of information exchange demands a standard external data representation. One such standard, Abstract Syntax Notation One (ASN.1), is defined by the International Organization for Standardization (ISO) and its transfer syntax, the Basic Encoding Rules (BER). This project presents the design of a C++ class hierarchy modeling the concepts presented in ASN.1 and encapsulating the BER encoding rules. The goal of this class hierarchy is to utilize the rich features of object-oriented paradigm to provide a clean and simple interface for OSI presentation layer protocols to application developers and implementors. This report discusses the class hierarchy in detail and shows how the primitive and constructed ASN.1 types are defined, encoded and decoded using this hierarchy.

Acknowledgements

This project is dedicated to my mother Mrs. Raj Khera and my father Mr. Ramesh Khera for their persistent encouragement, support, motivation and guidance.

I would like to express my sincere gratitude to the Chairman of my Committee, Dr. Dennis Kafura, for his precious and ubiquitous guidance, and bearing with me in all the adverse circumstances. Sincere thanks to my comittee members Dr. Sallie Henry and Dr. J.A.N. Lee for their valuable suggestions and serving on my committee.

I would also like to thank my sister Rani Khera for encouraging me, Stacey Blachke for helping me edit this report, and my other friends who always motivated me for working towards this project.

Table of Contents

1. Introduction 1

2. Why this class hierarchy ? 1

3. The Presentation Layer 2

3.1 Abstract Syntax 3

3.1.1 Abstract Syntax Notation One (ASN.1) 4

3.1.1.1 TYPES 5

3.1.1.2 VALUES 6

3.1.1.3 MACROS 6

3.1.2 Transfer Syntax Notation 6

4. Background 8

4.1 ISODE 8

4.2 CASN1 11

4.3 OSTO 11

5. The Class Hierarchy Design 11

5.1 Primitive Types 13

5.2 Constructed Types 15

6. Using and Testing Classes 20

7. Limitations and Future Work 24

8. Conclusions 24

References 25

List of Figures

Figure 1. The three field of the BER encoded ASN.1 type 7

Figure 2. Conversion of concrete data-structures to octets 7

Figure 3. The Communication Infrastructure 10

1. Introduction

In the present information age, information is as important to an organization as its employees or equipment. The structure and transfer of information are of key importance to anyone who uses this information. Computers, the tools used to manipulate information, have become commonplace. Networks, which interconnect computers, help move information and thus have revolutionized the use of computers themselves. However, physically connecting computers is not enough to achieve mobility of information, a language needs to be defined which allows these interconnected machines to communicate. This necessitates the definition of a 'protocol' and the common framework called 'protocol suites', under which the protocols are administered. The two best known protocol suites are the Internet suite of protocols, sponsored by the United States Department of Defense (USDOD), and the Open System Interconnection (OSI) suite, sponsored by the International Standards Organization (ISO).

2. Why this class hierarchy?

Writing applications that are able to work on multiple computers using either of the above protocol suites, is not an easy task; data has to be shared, complete integrity of the shared data must be ensured and the interface has to be defined. All of this entails an enormous amount of detail some of which is relevant to the application and other of which is largely irrelevant. In order to hide unnecessary details from the application developer, a "good" interface has to be designed which encapsulates the protocol details and provides a higher level of abstraction. This project's main focus is to implement such an encapsulation. The properties of encapsulation and data abstraction are met by the object-oriented paradigm; thus this project: The Design of C++ Class Hierarchy Modeling the Concepts Presented in Abstract Syntax Notation One (ASN.1) Encapsulating the Basic Encoding Rules (BER). Its prime objective is to provide a clean and simple interface to application developers and implementors, thus hiding the details involved with encoding and decoding the data types to be transmitted over the network.

The C++ class hierarchy resulting from this project is important because of the following reasons. First, it will ease the development process for the distributed applications. A good interface would encourage the development of new applications, by avoiding awkward interface with a steep learning curve. Second, because of the clean interface and other object-oriented concepts of C++, the code will be much more maintainable and reusable, which offer more returns for less effort. Third, since the class hierarchy presented here encapsulates the OSI's Presentation Layer Protocol, it will ease access to the ISO's OSI upper layer network protocols.

3. The Presentation Layer

Since the Presentaion layer is the one encapsulated by the C++ classes designed in this project, it requires some discussion. The Presentation layer deals with information representation issues such as encoding data, data compression and cryptography. The Presentation layer entities convert machine dependent complex data structures into octet strings which are then transferred by the lower layers. Peer application processes exchange information, through the use of presentation services, as an *information unit*. The Presentation layer is responsible for representing (encoding) the contents of an *information unit* and uses the following three forms to do so:

Source : This form represents data on storage media on the transmitting end systems.

Destination : This representation form is used by the receiving end system to store the information in its own storage media.

Transmit : This representation form is understood by both connected systems and used while the information is being transmitted over the session connection.

It is the responsibility of the Presentation Layer to manage the appropriate transformation performed on the information unit during the course of information transfer. The following two transformations: 'Source --> Transmit' and 'Transmit --> Destination' are performed at the transmitting end and the receiving end, respectively. The syntax known as an 'Abstract Syntax,' formally defines the

contents of an information unit without reference to any encoding technique that may be used to 'store' such a unit. The transmit encoding, which is used to encode the information in a precise representation form understood by the peer presentation entities, is commonly known as 'transfer syntax'. By using this transfer syntax, the peer presentation entities can exchange information over a session connection.

Presentation services are provided in many different ways, of which the most common are the following. First, Sun Microsystems' External Data Representation (XDR) standard, which is a library of routines a programmer can use to convert data types from abstract syntax to transfer syntax (this is discussed in the following section). The functions provided in the library are capable of encoding and decoding primitive data types e.g. Integer, Boolean, Real etc. The application developer is responsible for complex data types like 'C' structures by calling individual functions for each primitive type which comprises the complex type. Second, there is the Presentation Service described in this project, which is used by ISODE [ROSE91]. It provides a Data Description Language (Abstract Syntax Notation One in this case, discussed in a later section). ASN.1 is capable of describing data structures that are independent of the hardware and the operating system being used. These data descriptions are then translated to a programming language of the user's choice so that the translated data descriptions can work on a given computer system. The code is then generated by the application developer to encode and decode the data types needed by the application, using the encoding rules (BER in this case). The user is responsible for learning the Data Description Language and writing the description of the data.

3.1 Abstract Syntax

Abstract Syntax describes the structure of the units of data that are exchanged over the network. In a distributed application, the data which is exchanged over the network must be identically understood by the sender and the receiver. To accomplish this, the specification of the protocol used by the distributed application first defines each data type using an *abstract representation,* i.e., each data type is described independently of machine-oriented structures and restrictions. The protocol specification then defines the

concrete representation equivalent to the abstract data type by using any high-level programming language that is native to a given machine. When data is transmitted, two mappings must be performed:

- the data structure is mapped to the abstract syntax for the data type; and
- a transfer syntax is applied to the abstract syntax to obtain the value corresponding to the data type to be transmitted.

At the receiving end, inverse mappings are applied in the reverse order so that the data transmitted is converted back to its original form.

ISO has developed and standardized a language for the specification of 'abstract syntax', called the Abstract Syntax Notation One (ASN.1) [ISO 8824]. There is also a set of Basic Encoding Rules (BER) [ISO 8825], which constitute a 'transfer syntax' for information units that can be expressed by ASN.1. These two specifications are discussed below.

3.1.1 Abstract Syntax Notation One (ASN.1)

The task of Abstract Syntax Notation One is to describe data types in a machine independent fashion. ASN.1 has a rich syntax for describing data types and provides a powerful macro facility for extending its grammar. The following is a brief overview of ASN.1 syntax and its semantics.

ASN.1 has a set of primitive data types and also provides a facility to construct new elements with these primitive types. In ASN.1 there are four types of lexems:

1. Words: consists of upper-lower case letters, digits, and hyphens, starting with a letter, e.g., 'A123,' 'a321-12a'.
2. Numbers: consists only of digits, e.g., '123', '321'.
3. Strings: can be character, hexadecimal, or binary strings, e.g., 'character string', '0123456789ABCDEF'H, and '01'B respectively.
4. Punctuation: gives readability to the ASN.1 description.

Comments begin with character sequence '--' and continues until end of the line or until another '--' is encountered. A collection of ASN.1 descriptions, relating to a common subject, is called a *module.* A typical module looks like:

<<module>> DEFINITIONS ::=

BEGIN

<<linkage>>

<<declarations>>

END

The <<module>> term names the module, the <<linkage>> term relates this module with other modules, and lastly, the <<declarations>> term contains the actual ASN.1 definitions. Entities defined in ASN.1 can be one of the following three kinds: *types, values,* and *macros* which can be named using an ASN.1 word discussed above. Words used for types, values, and macros can be differentiated using the following case conventions. When it is a:

- *type*, the word starts with an uppercase letter (e.g., Name)
- *value* (an instance of a type), the word starts with a lowercase letter (e.g., id-pt-delivery)
- *macro*, the word consists entirely of uppercase letters(e.g., SIGNED).

3.1.1.1 TYPES

The following types are defined in ASN.1:

- Simple Types like Boolean, Integer, Real, Bit String, Octet String and Null;
- Object Types like an Object Identifier and an Object Descriptor;
- Constructed Types like Sequence, SequenceOf, Set, and SetOf;
- Tagged Types belonging to any of these following four classes of tags Universal Tags, Application-Wide Tags, Context-Specific Tags, and Private-Use Tags;

• Meta Types like Choice, Any, and External.

All of these types can be assigned values using the notation provided by the ASN.1 language. A detailed discussion of these types can be found in [ROSE90], [GAUD89] and [CHAP86].

3.1.1.2 VALUES

ASN.1 specifies a way for describing values, which are human-readable descriptions of the data-type encoding, used mainly by programmers to develop and debug applications.

3.1.1.3 MACROS

ASN.1 also defines a Macro notation which allows the grammar to be extended to meet the requirements of the abstract syntax designer. It is different from the macro utility of some other programming languages where only textual substitution occurs and the parsing remains unchanged; in ASN.1 the macro utility literally rewrites the grammar rules of the ASN.1 language. The high level definition of macro is:

<<macro>> MACRO ::=

BEGIN

TYPE NOTATION ::= <<type syntax>>

VALUE NOTATION ::= <<value syntax>>

<<supporting syntax>>

END

where <<macro>> gives the name to the macro.

3.1.2 Transfer Syntax Notation

The task of the transfer syntax notation is to represent unambiguously the values of data types that are transmitted on the network. The OSI transfer syntax notation is the Basic Encoding Rules (BER). Using

the BER, instances of ASN.1 types are encoded as a stream of octets. The BER uses a "TLV" approach to mapping between the abstract syntax and the concrete syntax, where each data type is encoded as a **T**ag, a **L**ength, and a **V**alue. The Tag field corresponds to the tag defined by the data type's abstract syntax and is encoded as one or more identifier octets. The Length field normally indicates how many octets are used for the encoding of the value portion of the data type and is encoded as one or more length octets. The Value field of the data type is encoded as zero or more content octets. Thus, a ASN.1 type is encoded in three fields as shown below in Figure 1, each of these fields can be of variable length.

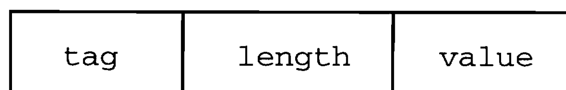


Figure 1. The three fields of BER encoded ASN.1 type.

Finally Figure 2. summarizes how data definitions can be converted first to Abstract syntax and then to Transfer syntax.

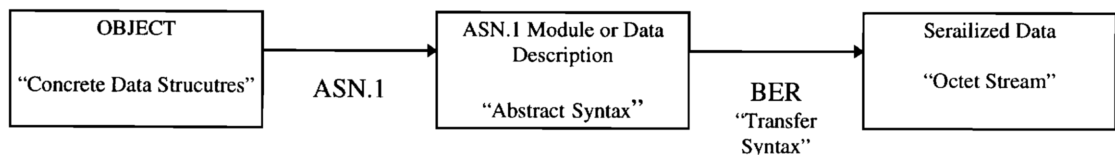


Figure 2. Conversion of concrete data-structures to octets.

4. Background

The underlying communication infrastructure used in the project is based on the ISO Development Environment (ISODE) [ROSE91], a non-proprietary implementation of the upper layers of OSI protocols. The purpose of the project is to utilize various features of object-oriented programming like encapsulation, data abstraction, and polymorphism to hide the complexity of the OSI protocols from the application developers so the formidable power of OSI may become available through a clean and simple interface. This work involves designing, developing and testing the class hierarchy for various types defined by the ASN.1 and developing the encoder and decoder interfaces that aid in developing a clean and simple interface thus accomplishing the basic objective of the project. Although the actual implementation of the encoder and decoder interface is tailored for the ASN.1 compiler and BER of ISODE [ROSE91], it can act as a front end for many of the different ASN.1-C/C++ compilers which are available, with only minor changes in the design. This section reviews various articles and texts that were the main sources of information and background reading needed for this project.

4.1 ISODE

The main motivation behind this project is the technical report [LAVE91] which discusses the synergy between object-oriented programming and the Open System Interconnection model and can be considered as a follow-up work to this report. It discusses "how the synergy between OSI and object-oriented programming can be used, first, to empower the object-oriented paradigm by infusing it with the distributed programming capabilities inherent in open systems communication, and second, to harness the power of an open system by organizing its services within an object-oriented language framework, thus improving the software engineering practice of building distributed object-oriented applications." [LAVE91]. The report also discusses the concept of structured communication showing the two points of synergy between the structured communication and the object-oriented paradigm which directly applies to

the project. First, ASN.1 has the representational power to express the strongly-typed method signatures found in typical class definitions, and second, the encapsulation properties of objects allow the translation mechanism implied by ASN.1 to be concealed from the user of these objects. Thus, these structuring facilities of object-oriented programming can be used to control the complexity of the protocol implementation.

The communication infrastructure used in the report [LAVE91] is also based on the object-oriented re-engineering of the ISO Development Environment (ISODE). It discusses the advantages of applying an object-oriented model to the ISODE as follows: reduction in the complexity of the service access point interfaces and protocol machine structure at each layer, encapsulation of the state of upper layer machine protocols, higher performance through inter-layer function inlining, and enhanced support for concurrent operations and asynchronous communication.

Looking at the communication infrastructure (figure 3), the aim of this project is to make available a collection of classes for encoding and decoding the ASN.1 types which have been previously implemented and made available for reuse by providers. The classes will be defined in a programming language-independent style. The provider of the classes implements a class in the language of his or her choice. These classes can be used then by the application developers as an interface to the underlying BER to define, encode and decode his or her types.

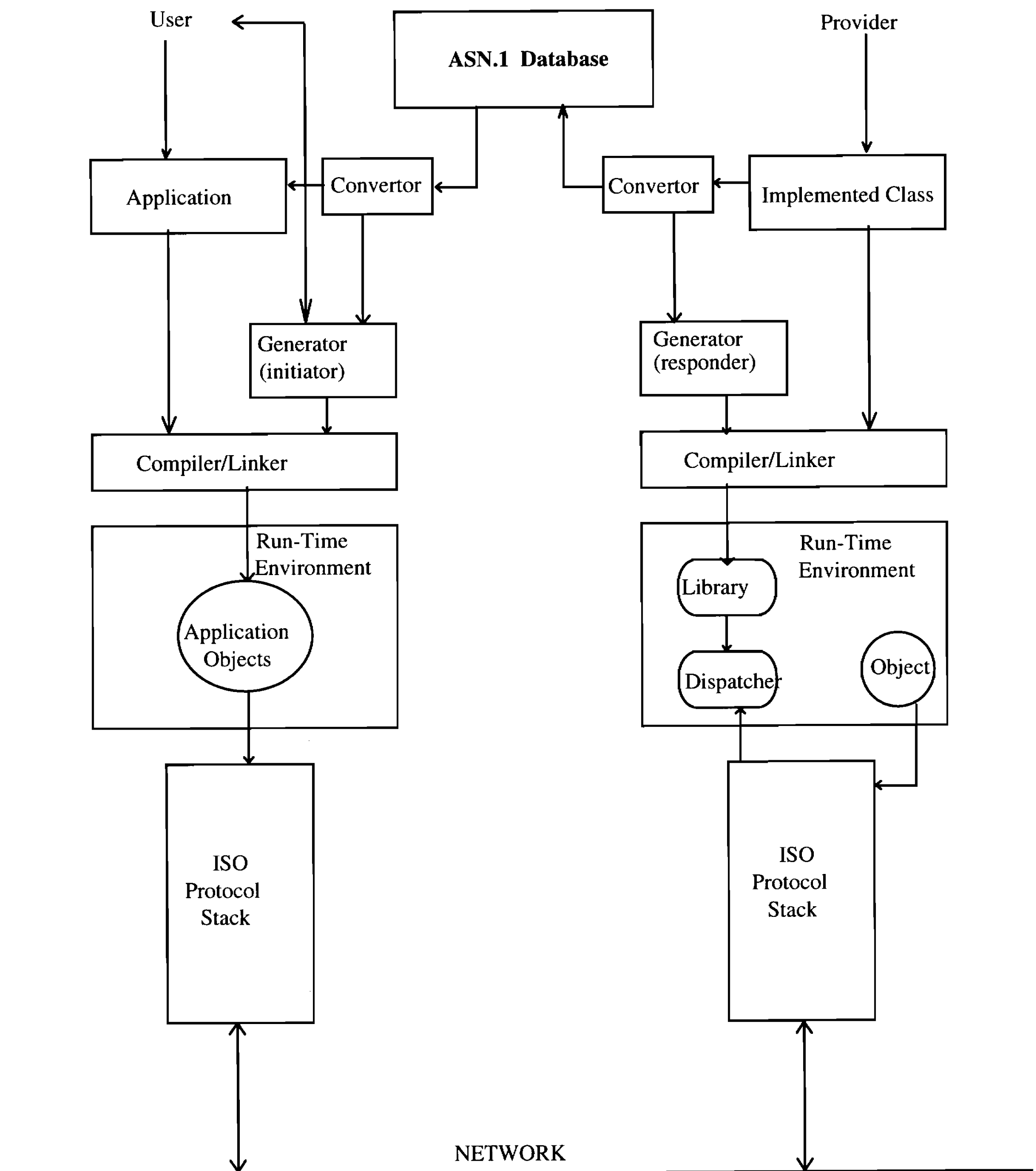


Figure 3. The Communication Infrastructure.

4.2 CASN1

The paper "The Design and Implementation of an ASN.1-C Compiler" [NEUF90] discusses an ASN.1-C compiler, CASN1, its design issues and its use of the BER to encode and decode the data types. The main design goals of the system were to make encoding and decoding complex types efficient and to free implementors from the oppressive work of translating protocol-defined data types and building their associated encoders and decoders.

4.3 OSTO

OSTO [KOI90] is both a model and a set of software tools for developing portable communication protocols and distributed applications. It consists of a C++ class library and a compiler. The CLASN [REIL] compiler is used to generate Abstract Interface Classes and Parameter-type Classes from an ASN.1 protocol data unit and Service Primitive Definitions both written in ASN.1, which are input to an OSTO compiler with user-written concrete protocol classes to produce the C++ implementation of desired protocols, message classes and support of distribution. These classes, when compiled on a C++ compiler produce an OSI protocol stack implementation for transferring data between two machines.

5. The Class Hierarchy Design

As stated before, the objective of the project is to utilize the rich features of object-oriented paradigm to develop a clean and simple interface for OSI presentation layer services. The initial work of this project involved, studying how conversions of ASN.1 types take place in ISODE[ROSE91], and understanding the intermediate internal data structures used in this conversion. These intermediate internal data structures act as an internal interface to ISODE's encoders and decoders. When the work on this project started, the ISODE 6.0 had just been released and it had three phases for generating the final C code from ASN.1 data descriptions; namely the ROSY (Remote Operation Stub generator, Yacc based), PEPY

(An ASN.1 compiler) and POSY (Pepy Optional Structure generator, Yacc based). Some changes were made to the ISODE in later versions; a new table-driven approach was added with the new release i.e., version 6.14, and POSY and PEPY were replaced by PEPSY. These tools produced enormous amount of C-code which was complicated to use and would in turn made the application development extremely challenging. For this reason the primary aim of the project was to eliminate or encapsulate this intricate C-code.

The design phase of this project started with the design of the class structure for the primitive ASN.1 type INTEGER. The idea was to design a class in such a way that one should be able to encode and decode ASN.1 types merely by calling a function encode() or decode() without concern for how the encoding or decoding is achieved. It was designed in such a way that this interface of encoding and decoding was kept common across all the ASN.1 types. To illustrate this idea, let us assume that 'intvar' is declared as an INTEGER. The statement 'intvar.encode()' should be enough to encode it, and similarly 'intvar.decode()' to decode it. This was accomplished, after several design changes, by utilizing several of the features that an object-oriented programming language provides for data abstraction, polymorphism, and encapsulation. The final design is explained in the following discussion.

The 'AsnType' class, which is the parent class to all ASN.1 types, contains the components that every other class inherits. The class declaration looks like:

```
class AsnType {
public:
        State state;                  // Enumerated constant associated with class
        Tag   tag;                    // Object of Class Tag
        AsnType() {};                 // Default constructor
        AsnType (Tag t, State s = UNDEFINED_STATE ): tag(t), state(s) {};
        virtual Tag myTag() { return tag;}
        virtual  PE encode() = 0;
        virtual void decode(PE) = 0;
};
```

There are two data members in 'AsnType' class. The first data member is the State 'state' of the object, can take one of the three possible values: DECODED_STATE, (when the object is in its primitive form), or ENCODED_STATE, (the object has been converted to a presentation element PE, ready to be sent over the network), or UNDEFINED_STATE. The UNDEFINED_STATE was added to solve some issues related to constructed types like SEQUENCE and SET and is discussed in a later section. The second data member is the Tag 'tag', which contains the information about the kind and value of the tag associated with any particular object and is itself defined as a class. The member function myTag() returns the tag value, i.e., the class and the value associated with any particular object. The constructor AsnType(Tag t, State s) construct the 'AsnType' object with Tag 't' and State 's'.

5.1 Primitive Types

The primitive types like Integer, Real, Boolean, OctStr BitStr, IA5String, and GraphStr are designed as follows: The class design for each of the primitive types is very similar. This simplifies understanding the design, thus making the classes easier to be used by the application programmers. All primitive-type classes are derived from the class AsnType discussed above and add various details required by that particular type. Below is an example of actual implementation of one of the primitive-type classes:

```
class INTEGER : public AsnType {
        int length;
        union {
          int x;                  // holds the value in decoded state
          PE pe;                  // hold the value in encoded state
        };
public:
        typedef int BASE;
        static Tag TypeTag;       // initialized to Universal class with tag of 2.
        INTEGER(void);            // constructor creates undefined state object
        INTEGER(int x);           // creates an decoded state object
```

```
        virtual ~INTEGER():        // destructor
        PE encode();               // the encoder for integer type
        void decode(PE);           // the decoder
        operator int ();           // unary operator 'int'
        int operator = (int);      // binary assignment operator
};
```

The INTEGER class contains a variable 'x' of integer type to hold the value of the object in its decoded state and 'pe' a pointer to a PE type, to contain the encoded value of the object. These two variables are implemented as a C union so that only one of them exists at any given time, i.e., a object can be either in encoded or decoded form at any given time, which is maintained by the 'state' variable of the AsnType class, which the INTEGER class inherits. The INTEGER class also has two constructors. The first constructor takes an integer as a parameter to construct an object in the DECODED_STATE and the second constructor constructs an INTEGER in UNDEFINED_STATE. It also has a static variable 'TypeTag' which is initialized as a UNIVERSAL_CLASS tag with the value of 2. It has 'PE encode()' and 'void decode(PE)' functions defined which are used to convert the object from one state to another. The encode() member function encodes an integer in the decoded state, to convert it into a PE 'pe' using the underlying ISODE routine 'int2prim' changing the state of the object to ENCODED_STATE. If the data is already in the encoded state it returns an error. The decode() member function decodes the object in the ENCODED_STATE. It takes PE 'pe' as a parameter and converts it to an integer using the ISODE routine 'prim2num' and changes the state of the object to decoded state. It also has an operator 'int', which returns the value of the object when it is in DECODED_STATE, and an assignment operator '=' to assign an integer value to the object. All other primitive-type classes are constructed in a similar fashion with different TAG values and use different underlying ISODE routine in the encoder and decoder.

5.2 Constructed Types

Primitive-types were simple to design since there were many similarities among them and there was almost a one-to-one mapping between them and corresponding C-types. However, there are more complicated issues involved in the design of classes for constructed-types, since constructed ASN.1 types are used to represent complex data types formed by combining the primitive data types. Below is discussed the class design for SEQUENCE, one of the constructed ASN.1 type, which covers most of the concepts used in developing the class hierarchy for the constructed type.

The ASN.1 type SEQUENCE, is used to represent an ordered list of zero or more elements; for this reason the class design used to represent the Sequence type uses a linked list to represent an ordered list of zero or more elements in the easiest possible way.

The SEQUENCE class is implemented as a class which derives from AsnType in the same way as the primitive types. It contains the data member of type PE to hold the value when in encoded state, and a linked list of SequenceElement which is implemented as a class 'SequenceList' and an integer variable 'current' to keep track of the element in the sequence which is being processed. The declaration of SequenceList class is as follows:

```
class SequenceList {
     SequenceElement* first;
     SequenceElement* last;
     SequenceElement* current_element;
public:
   SequenceList() {first=last=NULL_ELEM; return;}
   ~SequenceList(){};                //need to add destructor here
   void add(SequenceElement*);
   void start() {current_element = first; return;}
   int  done() {return(current_element == NULL_ELEM); }
   void next();                      // advances the pointer to current elem.
   SequenceElement* current() {return current_element;}
```

```
    int get_tag();                    // gets the tag of the current element
    int get_state();                  //gets the state of current elem'
};
```

It has the following data members: SequenceElement* 'first' which points to the first element in the list, 'last' which points to the last element in the list, and 'current_element' which points to the current element being processed in the linked list. It also has a constructor SequenceList() which initializes the data members 'first' and 'last' to NULL_ELEM, which is a (SequenceElement*) typecast to a NULL pointer, and the destructor ~SequenceList(). The SEQUENCE class has the following public member functions: void add(SequenceList*) which adds a SequenceElement to the end of the SequenceList; void start() which sets the current_element data member to point to the first element in the SequenceList; int done() which returns true if the current element points to the NULL_ELEM which means the end of the SequenceList has been reached or all of the elements in the SequenceList have been processed; void next() sets the current_element to point to the next element in the SequenceList; SequenceElement* current() returns the pointer to the Sequence Element pointed to by the data member current_element in the SequenceList; int get_tag() returns the tag value of the item pointed to by the current_element; and int get_State() which returns the state of the item pointed to by the current_element.

SequenceElement, of which the SequenceList is composed, is implemented as a class defined as follows:

```
class SequenceElement {
public:
  AsnType*  element;
  SequenceElement* next_elem;
  SequenceElement(AsnType* asnvalue) {
     element = asnvalue;
     next_elem  = NULL_ELEM;
  }
```

```
  SequenceElement* next() { return next_elem;}
  void next(SequenceElement* after);
  PE encode();
  void decode (PE pe) {};
};
```

SequenceElement has a data member 'element' which is a pointer to an AsnType; next_elem which is a pointer to the object of SequenceElement and is initialized to NULL_ELEM; a constructor SequenceElement, which takes a pointer to an AsnType and makes the data member 'element' point to it and also sets the data member 'next_elem' to NULL_ELEM which is #defined as (SequenceElement* 0); the member function SequenceElement* next(), which returns the pointer to the element pointed to by the 'next_elem'; the overloaded member function void next (SequenceElement*) which makes the next_elem point to the parameter being passed; the member function void decode(PE) is a null function; and PE encode() calls the encoder for the element it's called on.

The SEQUENCE class is defined as follows:

```
class SEQUENCE : public AsnType {
        PE  pe_seq;                // to hold the object in encoded state
        int current;               // points to the element being processed.
        SequenceList  sequence;    // Linked list
public:
        static Tag TypeTag;        // Hold the Tag value for the Seq
        typedef int BASE;
        SEQUENCE (BASE v):(ASN_SEQUENCE_TAG, UNDEFINED_STATE);
        SEQUENCE () : (ASN_SEQUENCE_TAG, UNDEFINED_STATE);
        ~SEQUENCE () ;
        BASE operator=(BASE s) {return 0;}
        PE encode();               // Seq Encoder; explained below
        void decode(PE);           // Decoder ; explained below
        void add(SequenceElement* element);
```

```
};
Tag SEQUENCE::TypeTag = Tag(UNIVERSAL_CLASS, 16);
```

The SEQUENCE class has the public members static 'TypeTag' of type 'Tag' which holds the Tag value of UNIVERSAL_CLASS with value 16, as discussed above; and this is defined as standard TAG for the Sequence. The constructor, SEQUENCE(), initializes the TAG value given for every AsnType to ASN_SEQUENCE_TAG, which is defined the same as the TypeTag discussed above and the state to UNDEFINED_STATE.

The member function PE encode() is the one which manages the mechanics of encoding all the elements in a sequence and thus hides the unnecessary details from the user. This is accomplished in the following steps: first, a data structure is allocated using the underlying ISODE routine pe_alloc(...) and the data member 'pe_seq' is set to point to the allocated structure; second, the SequenceList member function 'start()' is called to initialize the sequence i.e. the data member current_element now points to the first element in the sequence; third, start with the first element in the SequenceList by calling the member function 'current()' of the SequenceList class which returns a pointer to the element pointed to by the current_element, which is pointing to the first element as discussed above; fourth, the encode function of the element is called which because of the polymorphic nature of C++, maps to the encode function of the particular type of the element of the sequence. It then adds the PE returned by the encode function to the 'pe_seq' using the ISODE routine seq_add() to build the encoded sequence. If there is some error while encoding the sequence-which means the element expected was not present-it is handled appropriately depending on whether the element was optional or mandatory. This will be discussed in detail in the following sections. Once the element is encoded and added into the sequence, the 'next()' function is called to increment the pointer to the next element in the sequence and the whole process in repeated until the end of the sequence is reached. Then the encode function returns the pointer to the resulting 'pe_seq' which contains all the encoded elements of the sequence.

The void decode(PE) member function decodes the encoded sequence with a mechanism which is similar to that of encode(). It starts with the PE pointing to the Sequence and the actual Sequence itself, it then decodes each element after matching them in both the lists. It starts by calling start() similar to the encoder and works on each element thereafter. It extracts elements from the passed PE using the underlying ISODE function seq_find() and, if it is not a NULLPE, it checks to see if its TAG number and class matches with that of the one in the sequence data member. If they match, the element is decoded and its value is assigned to the element in the sequence and the next PE is looked for in the sequence PE and matched with the corresponding element in the sequence until the entire sequence has been completed.

Another feature of the sequence which was discussed above is the attribute of an element in a sequence to be optional. An optional element in a sequence is defined as an 'Optional' class template object. Below is the listing of this class.

```
template< class T> class Optional : public T {
public:
    PE encode();                    // Optional item encoder;
    Optional() : T() {};            // Default constructor
    Optional(T t) : T(t) {};
    T operator=(T t) { return T::operator=(t); }
};
```

This class template takes any ASN.1 type, declared and defined in the file AsnTypes.h, and affects the behavior of encoding the Sequence elements when encoding that particular Optional element. If the element in the Sequence is defined as optional, and is not present, the encoding routine returns to the Sequence encoder a value NULLPE of PE, which is interpreted in the Sequence encoder as an Optional element not present, thus differentiating between Optional and Mandatory (not optional) elements.

The desing of the constructed type SET is very similar to the SEQUENCE class. Other constructed types like SequenceOf, SetOf etc. are also designed with the same design philosophy discussed in the beginning of section 3.

6. Using and Testing classes

This class hierarchy to model ASN.1 data types can be used in the following way. An object of the class is created, for the ASN.1 type being used or tested, then it is encoded and decoded using the encode() and decode() member functions of the class for that particular ASN.1 type. The PE returned by the encoding function can be transmitted over the network and decoded by passing it as a parameter to the decode() function and the information retrieved from it. For testing purposes, the function 'invoke(PE)' was designed. It prints, the TAG (the class and the value), the LENGTH, (the number of octets needed to transmit the object), and the VALUE of the encoded object, the **TLV** approach discussed in section 3.1.2. The invoke(pe) function uses the underlying ISODE routines ps_get_abs(PE) to check for the octets needed to transmit the variable over the network; ps_alloc(std_open) to allocate the std_open Presentation Stream; and pe2pl(ps,pe) which converts a presentation element to a presentation list which is a human-readable, and unambiguous way of describing a presentation element. It prints all the necessary information to check whether the encoding was done correctly. Thus, if everything is encoded properly, the invoke(pe) function would succeed; it would print the number of octets needed, the ASN.1 type, i.e., TAG, and the value of the encoded variable; then it would return normally, or else, fail. Testing continues by decoding the 'pe' that was returned by the encoder and used by invoke to give the original values used before it was encoded, which should match with the values output by invoke(pe) function, and printed using standard C/C++ output functions. Thus, if this two-step testing process is successful, it proves that the class as well as its encoder and decoder design is valid. This process is illustrated by the following piece of code used to test the INTEGER type:

```
#include "Integer.h"        // Class design for the INTEGER
#include "Invoke.h"         // Function used for testing.
#include <stream.h>
main() {
    INTEGER x(6);
    cout << "value of x is: " << (int)x << "\n";
    PE encoded_X = x.encode();
    invoke(encoded_X);
    INTEGER z;
    z.decode(encoded_X);
    cout << "value of z is: " << (int)z << "\n";
};
```

First an object of INTEGER type is created and then it is encoded into a Presentation Element(PE) using the encoder function of the INTEGER class. Then the invoke() function is called on this PE to check whether the encoder encoded the INTEGER properly. Finally, the PE is decoded using the decode() member function of the INTEGER class and the value is printed from the resulting decoded value to match it against the original value. The result of executing the above piece of code looks like:

```
value of x is: 6
Octets needed: 3
( UNIV INT 0x1
  0x06
)
value of z is: 6
```

A similar process is used to test all the primitive types like Real, Boolean, Octet String, Bit String, IA5String, etc. A slight variation is needed to test the constructed types like Sequence, Set, SequenceOf and SetOf. In each case we have to derive a class from the class that represent that particular type. Let's take the

example of a Sequence. Below is the ASN.1 data definition of Passwd which is defined as a sequence of three different ASN.1 types. Following the data definition is the C++ class generated for it and then the code which will be required to define an instance of this class and use or test it.

```
PasswordLookup DEFINITIONS ::=
BEGIN
-- types
Passwd ::=
        SEQUENCE {
          name OCTSTR,
          passwd IA5String OPTIONAL,
          quota INTEGER DEFAULT 0,
        }
END
```

```
class Password : public SEQUENCE {
public:
        OCTSTR name;
        Optional<IA5STR > passwd;
        INTEGER quota(0);
        Password() {
          MEMBER((&name)); MEMBER((&passwd)); MEMBER((&quota));
        }
};
```

```
void Seqtest()
{
        Password MyPassWord;

        MyPassWord.name = "Rajesh";
        MyPassWord.passwd = "missing";
        MyPassWord.quota = 6;

        Pe MyPe = MyPassWord.encode();
        invoke(Mype);

        PassWord NewPassWord;
        NewPassWord.decode(Mype);


        cout << " The name field is :" << (char*)NewPassWord.name;
        cout << " The passwd field is :" << (char*)NewPassWord.passwd;
        cout << " The quota field is :" << (int)NewPassWord.quota;
}
```

A class for a variable of a type SEQUENCE is constructed by deriving the class from the SEQUENCE class. The expression MEMBER((&i)) uses the following macro to add an element to a sequence.

#define MEMBER(x) add(new SequenceElement x)

The object, thus initialized, is then encoded using the encode() member function as discussed above. After this, a procedure similar to the primitive types is followed. The PE 'pe' returned by the encoder is passed to the invoke() routine to print out its elements to check to see whether the elements were correctly encoded. After that, the 'pe' is decoded using the SEQUENCE member function decode() and then individual elements of SEQUENCE are printed to check to see whether the elements decoded match the original elements that were used to construct it. The attribute of an element in a Sequence of being OPTIONAL is handled by the Optional template class which can be used to declare a sequence element as:

Optional<IA5Str> passwd;

Even a default value can be specified by adding it at the time of declaration as follows:

INTEGER quota(0);

Similarly, the attribute of a Sequence element of being Implicitly or Explicitly tagged can be handled by just declaring an object of the Implicit or the Explicit class template. For example, if a variable of type REAL is to be used with implicit APPLICATION wide tag of value 0, it can be defined as:

Implicit<REAL, APPLICATION_CLASS, 0> My_imp_real_var;

or if this is an optional element, it can be combined with the above Optional class template class declaration as follows:

Optional<Implicit<REAL, APPLICATION_CLASS, 0> > my_var;

which gives the elements of a Sequence a wide range of functionality.

The testing process for SET is very similar to the one discussed above for SEQUENCE but differs slightly for SetOf, SeqOf and Meta-type Choice.

7. Limitations and Future Work

The ASN.1 universe is much bigger than the fairly constricted focus of this project. Due to the limited scope and a lack of time, some of the ASN.1 types were not considered for class design but should be addressed. Types like Object Identifier, Object Descriptor, Enumerated, NumericString, PrintableString, TeletexString, VideoTextString, UTCTime, GeneralizedType, VisibleString, GeneralString and CharacterString, plus Meta types like ANY, due to the small scope of the project, were left out of the class design. The class design can be extended to cover these types.

Automated tools can be constructed to generate the classes for any set of ASN.1 data definitions. These classes can then be compiled and executed to transfer data over the network. Since this class hierarchy is built as a front end for the ASN.1 BER, it can be modified to work with any ASN.1 BER implementation other than the one used, the ISODE. Another option would be to build similar types of classes, modules, or structures for different high level programming languages like ADA, Modula, or SMALLTALK etc., which would allow users familiar with different types of programming languages to work transparently to the underlying ASN.1 BER.

8. Conclusions

This project is a start towards the goal to develop a class hierarchy which will model ASN.1 types as discussed in section 4.1. The main objective, "To utilize the rich features of object-oriented paradigm to come up with a clean and simple interface for OSI protocols implementation" was achieved after many design changes. Most of the ASN.1 types were successfully encoded and decoded using the class hierarchy, including some of the more difficult-to-implement features of the constructed and tagged types as explained in section 5.2 and section 6.

The classes which have been developed, present a clean and simple interface for application developers so they can encode and decode ASN.1 types without having to beware of the low-level details; also they make the encoding and decoding of complex types simple, and above all they can work with the same interface even if one chooses to have a different underlying ASN.1 BER implementation.

The complicated C-code which is generated intermediately by ROSY and PEPSY in ISODE, as discussed in section 3, has been eliminated.

Successful design of these classes also demonstrate the synergy between Object Oriented Programming and the Open System Interconnection put forward by the [LAVE91] report.

References

[CHAP86] " A Tutorial on Abstract Syntax Notation One (ASN.1) ", Omnicon Information Service 25, Omnicon, Inc., December 1986.

[GAUD88] " An Object-Oriented Model for ASN.1 ", Philip Gaudette, Steve Trus, and Sarah Collins, Proceedings of the First International Conference on Formal Description Techniques, Sterling, Scotland, 6-9 September, 1988, pages 121-134, North-Holland, Amsterdam, 1988.

[GAUD89] " A Tutorial on ASN.1 ", Philip Gaudette, Technical Report NCSL/SNA -- 89/12, May 1989.

[HENS88] "OSI Explained end-to-end computer communication standards", John Henshall and Sandy Shaw, Ellis Horwood Limited, 1988.

[KOI91] "Object-Oriented Approach to Distributed Computation", J. Koivisto, J. Malka, USENIX C++ Conference Proceeding, p. 163-178, Washington DC, April 91.

[LAVE91] "The Synergy Between Object-Oriented Programming and Open System Interconnection", R. Greg Lavender and Dennis Kafura, Technical Report 91 -31, Dept. of Computer Science, VPI & SU, November 22,1991.

[NEUF90] "The Design and Implementation of an ASN.1-C Compiler", Gerald W. Neufield and Yueli Yang, IEEE Transaction on Software Engineering. Vol. 16 No. 10 October 1990.

[REIL] "Generating Object-Oriented Telecommunication Software Using ASN.1 Descriptions", Juha Koivisto, and James Rielly, IFIP British Columbia Draft.

[ROSE90] "The Open Book: A practical perspective on OSI", Marshall T. Rose, Prentice Hall, 1990.

[ROSE91] " The ISO Development Environment User's Manual - Version 7.0 ", Marshall T. Rose, Julian P. Onions, and Colin J. Robbins. Vols. 1-5, X-Tel Services Ltd., Nottingham, July 1991.

[TANM88] "Computer Networks", 2nd Ed., Andrew S. Tanenbaum, Prentice Hall, 1988.