Separating Representation from Translation

Of Shared Data in a

Heterogeneous Computing Environment

by

Robert W. Mullins

Project 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

James D. Arthur

Scott F. Midkiff

September 1993
Blacksburg, Virginia

Separating Representation from Translation

of Shared Data in a Heterogeneous Computing Environment

by

Robert W. Mullins

Dennis G. Kafura, Chairman

Computer Science

Abstract

Data sharing among heterogeneous computers involves both a model to represent the structure of the data and a model to translate the data to achieve the sharing. The translation, defined by a transfer syntax and corresponding set of encoding rules, is necessary because heterogeneous computers typically use incompatible implementations of the basic data types. The central question considered in this work is how to separate the representation of shared data from its translation. Such separation is useful because different systems use different encoding rules. The solution developed is a modification of an existing C++ data representation class hierarchy modeled after the Abstract Syntax Notation One (ASN.1) language. The class hierarchy was modified so that all translation specific elements were factored out of the representation class hierarchy and placed into an encoding rules class hierarchy. The rules class hierarchy is structured in a way that allows additional encoding rule sets to be added easily. Encoders and decoders have been written for the Basic Encoding Rules (BER) defined by the OSI standards and Sun's

External Data Representation (XDR) standard, allowing data structures to be translated without support from vendor provided routines so the class hierarchy can be used on any system with a C++ compiler. The class hierarchy has interoperated with existing servers using Sun's Remote Procedure Call (RPC) protocol and the Basic Encoding Rules (BER).

Table Of Contents

List of Figures

1.0 Introduction

To manage the increasing complexity of more functional distributed applications, it is necessary to develop abstractions that encapsulate the details of data sharing and allow programmers to focus on the application being developed. The goal of most of these abstractions is to provide transparent access to data that is distributed across more than one machine as if the data were all located on the local computer. The C++ class library described in this paper provides a mechanism for representing data structures independent of the internal representation of any of the data types involved on a particular machine. The class library also provides methods for encoding data structures constructed with the types of the data representation classes using two well known standards. The unique aspects of the class library are that it does not require a new language or preprocessor for an existing language to build shared data structures, it allows the users to control the names of types and identifiers used for building shared data structures, and it allows data structures to be translated by either of the two implemented transfer syntaxes. Chapter 1 introduces and defines the term *shared data* and presents several abstractions used for sharing data and developing distributed

applications in a heterogeneous environment. Two previous approaches to providing data representation and translation services along with their drawbacks are described, followed by a justification for separating the representation of a data structure from its translation.

1.1 Shared Data

In general, the term *shared data* refers to data that is accessible by two or more processes. When writing an application that shares data, it is necessary to consider many details involved in the sharing of the data as well as the details of the problem being solved. To free application programmers from many of these details, it is helpful to develop an abstraction that provides a useful set of operations and hides the details of the data communication. Many such abstractions or models have been developed and the choice of which to use depends on the application being developed. Short descriptions of a few of the models are given below.

One popular abstraction for sharing data in distributed applications is the distributed database model. In this model, parts of the database can be located on different machines each of which runs a data manager process responsible for all reads

and writes of the data located on that particular machine. A transaction manager is responsible for obtaining permission to access the data required by the user application and communicating with the proper data managers to perform the reads and writes requested by the user program and ensuring that replicated pieces of the database remain consistent. A locking mechanism must be employed to ensure mutual exclusion of writes so that the consistency of the data is maintained [Maek87]. This model is well suited to large database applications such as banking and airline reservation systems.

Another model for sharing data is the tuple space abstraction in which the group of communicating processes all have access to a single tuple space [Carr89]. When a process wishes to share a piece of data, a tuple containing the information is written into the tuple space. If a process needs exclusive access to a data item, it reads and deletes the tuple from tuple space. Since the tuple is no longer in the tuple space, no other process can read that data until the process that holds the item writes it back to the tuple space. If a process needs to read an item, but does not require exclusive access, a read that leaves the data in tuple space is performed. Since the data item is still in the tuple space, other processes are free to access it. When the tuple space is spread across multiple machines, or if programming mechanisms exist for accessing a remote tuple space, it is clear that the tuple space model can be used for building distributed applications.

Another way that processes can share data is via shared memory, where multiple processes have access to the same shared memory area. Any data stored in the shared memory is accessible to all processes with access to that area of memory. To ensure consistency of a data item, a process must be granted shared access for a read or exclusive access to perform a write. Semaphores are the typical access control mechanism for shared memory. Clearly, since shared memory can be used to share data between processes on the same machine, if the shared memory is distributed, it can be used to share data among processes executing on different machines. Li and Hudak present an implementation of distributed virtual memory in [Li89] that allows multiple processors to have a copy of a given memory page for reading. If a process writes to the shared page, the memory manager must invalidate the copies of the page that exist on other machines. When a process attempts to read from a page that is not located on the local system or has been invalidated, a distributed page fault occurs and the memory manager must obtain a new copy of the page for that process. To the processes sharing data, the page of memory that contains the data appears to be a normal shared memory page. The fact that it is copied to and from the network is hidden by the memory manager.

Remote operations or Remote Procedure Calls (RPC) [Sun88] are another method for sharing data across a network. In this model, a program that wants to access data on a remote machine is known as a client or initiator, and the process that services client

requests is called a server or responder. A client assembles the arguments the server will need to service the request, sends a message to the server that invokes a function to perform the desired action, and blocks, waiting for the result. Upon completion of the request, the server returns the result and the client continues processing. Often, special compilers are used to generate the code to marshall the arguments and to send and receive the messages, which makes an RPC call look like a simple procedure call to the programmer writing the client, and preserves the function call programming paradigm with which most programmers are familiar.

It is important to note that an implementation of any of the models presented above will require messages to be passed across the network among the systems sharing the data. To pass messages between systems, it is necessary for the systems to have a representation of the message. Clearly, it is advantageous to be able to represent the message with an abstract syntax that is independent of the message's internal representation on any particular machine. Each system must then be able to generate a concrete representation for the message from the abstract syntax description. Each system must also be able to translate between its internal representation for the data types of its concrete representation of a message and a standard network representation for the data types involved. This project concentrates on two important aspects of all message passing models: the representation and the translation of the data being shared.

1.2 Previous Approaches

Assuming a reliable mechanism to transport a stream of data from one system to another, two main issues must be considered in order to share data between processes on different machines. First, what data is to be shared, and second, since machines may use different internal representations for the data types involved, how will that data be translated in order to be properly interpreted on the remote machine. Any general method for sharing data must provide a way to abstractly represent the data independent of the machine being used, and a standard way to represent the data when it is actually sent across the network. Methods for addressing these two issues are commonly referred to as *presentation services.*

The abstract representation of the data to be shared is often accomplished using a data description language which may be little more than the types and constructs available for building data structures in a particular programming language. This is the approach used in Sun Microsystems's External Data Representation (XDR) standard [Sun88] which uses the data structures of the C programming language. The Abstract Syntax Notation One (ASN.1) language developed by the International Standards Organization (ISO) is a language used for describing data structures that is not related to any particular programming language [Steed90]. The description of a data structure

is translated, either manually by a programmer or automatically by a compiler, into a data structure in the programming language being used. Sophisticated compilers may generate code to perform the translation of the data to the standard network representation.

When data is shared between systems with different internal implementations for the data types involved, the data must be translated from the sending machine's internal format to the receiver's internal format. Since an application can share data with multiple systems, it is undesirable to write specialized translation code for each type of system involved. Whenever a new type of system is added, the application must be modified to support translation to the new machine's format. An approach to solving this problem is to develop a canonical standard for representing the data types to be shared known as a *transfer syntax* [Rose90]. When an application wishes to share data, it represents that data in the standard format. Each system knows how to translate its internal representation to the standard representation when it sends data to another system, and how to translate from the standard to its internal representation when it is to receive data from another system. For example, the TCP/IP standard specifies that all integers used in the protocol will be represented in *big endian* format [Stev90]. Any system that uses little endian integers internally, such as a VAX, must translate integers to be sent on a TCP/IP network to big endian form. Systems receiving integers know that they are in big endian format and require no translation if their internal form for integers is big endian, or they translate them to little endian if that is their internal form for storing

integers [Stev90]. These standards for representing data are often referred to as encoding rules since they specify the rules that must be followed when representing the various data types.

Presentation services have been provided to programmers in two ways in the past. The first is to provide a library of routines that the programmer can call to translate a set of data types from the internal representation to the format specified by the transfer syntax. This is the approach taken by Sun when encoding data using the XDR standard. A library routine is called when encoding or decoding a built-in data type such as integer or real. If a more complex data structure is to be shared, such as a C structure, the programmer is responsible for writing code to make calls to the appropriate library routines for each member of the structure. Typically, a routine is written to encode and decode each particular data structure that will be shared in a given application. These routines are then called as needed throughout the application program.

A second method for providing data translation facilities to programmers is to define a data description language not specific to any programming language and employ translators to create code to perform the encoding and decoding in a particular language. A programmer describes a data structure in the data description language and code is generated to encode and decode the data. Since code is generated by a programming tool, this method usually forces an awkward naming convention on the data structures

used by the programmer [Kaf92][Bran92]. In addition, the programmer must now learn the syntax and semantics of the data description language, and how this relates to the programming language being used. This approach is used in the ISO Development Environment (ISODE) [Rose91a] where Abstract Syntax Notation One (ASN.1) is the data description language used by the programmer which is run through a tool that builds C language data structures and functions to encode and decode the data structures.

1.3 Justification for Separating Representation from Translation

Although a data description language compiler such as the ASN.1 compiler implemented in the ISODE may dictate the Basic Encoding Rules (BER) transfer syntax as defined by the ISO, the ASN.1 language definition does not make any such restriction. In fact, the transfer syntax may be negotiated at connection establishment time. Alternate syntaxes, such as Sun's XDR standard or a lightweight version of the BER syntax defined by [Huit89], are often preferable to the BER syntax since they generally execute much faster than the BER translation routines. These syntaxes are usually less flexible than the BER syntax, but in many cases they are robust enough to achieve the desired data sharing. For example, the BER syntax allows for arbitrarily large integers although most machines place a limit on the size that can be represented, typically 32 bits. Therefore,

a transfer syntax that always encodes integers using 32 bits does not place any significant restrictions on the integers shared among such machines.

It is desirable to separate the representation of the data from the transfer syntax that will be used to achieve the sharing of that data, if it will allow the choice of the most efficient transfer syntax for a particular data structure. The goal of this project is to demonstrate that such a separation is not only desirable, but possible. Since our approach separates the representation of a data structure from how it will be encoded, a data structure is constructed without concern for how it will be translated and the connection can be parameterized with the transfer syntax to be used, or the connection establishment process can negotiate the most efficient transfer syntax that can be used by both systems.

The remainder of this report is divided into five sections. First, background necessary for understanding how the separation was achieved and the classes are presented, along with a demonstration of interoperability. Next, the implementation section explains the structure of the class library and the reasoning behind many of the design choices made. Third, a simple comparison or the performance of our class library, Sun's XDR routines, and ISODE's BER routines is discussed. Fourth, some conclusions and directions for future work are considered. A user's guide is presented in Appendix

A and Appendix B contains the complete source code for many of the examples used throughout the paper.

2.0 Achieving Separation of Representation and Translation

This chapter introduces the class library developed for building shared data structures along with necessary background information. First, the *Open Systems Interconnect* (OSI) layered communication model is presented with particular emphasis on the presentation layer where data representation and translation services are provided. Arguments are given in favor of using an object-oriented approach to implement communication protocols in general and presentation services in particular. The class library is described and an explanation is given of how the separation of data representation and translation is achieved. Finally, interoperability is demonstrated by using the class hierarchy to build a client which successfully interacts with an existing RPC server.

2.1 Background

In this section the layered communication protocol model will be introduced, including the concepts of a service, service user, service provider, and a service access

point which are used throughout all layers of the model. Finally, a more detailed description of the presentation layer will be given.

2.1.1 Layered Model

To understand the organization of communication protocols, it is important to be familiar with the layered model of computer communication architectures [Stall87]. To make the task of communicating data between processes on different systems tractable, a communication architecture is usually designed in a layered or hierarchical fashion. Each layer has a well defined set of services that it provides, a defined interface that must be presented to the layer above, and a protocol for communicating with a process implementing the same layer on a remote system.

The seven layer *Open Systems Interconnect* (OSI) model is shown in Figure 2.1 along with a brief description of the functions performed by each layer. As shown in Figure 2.1, the layers of the OSI reference model can be partitioned into two categories, upper and lower layers. The first four layers, physical, data link, network, and transport, make up the lower layers, and provide end-to-end data transfer services [Rose90]. The

upper layers are comprised of the session, presentation, and application layers, which provide application-oriented services.

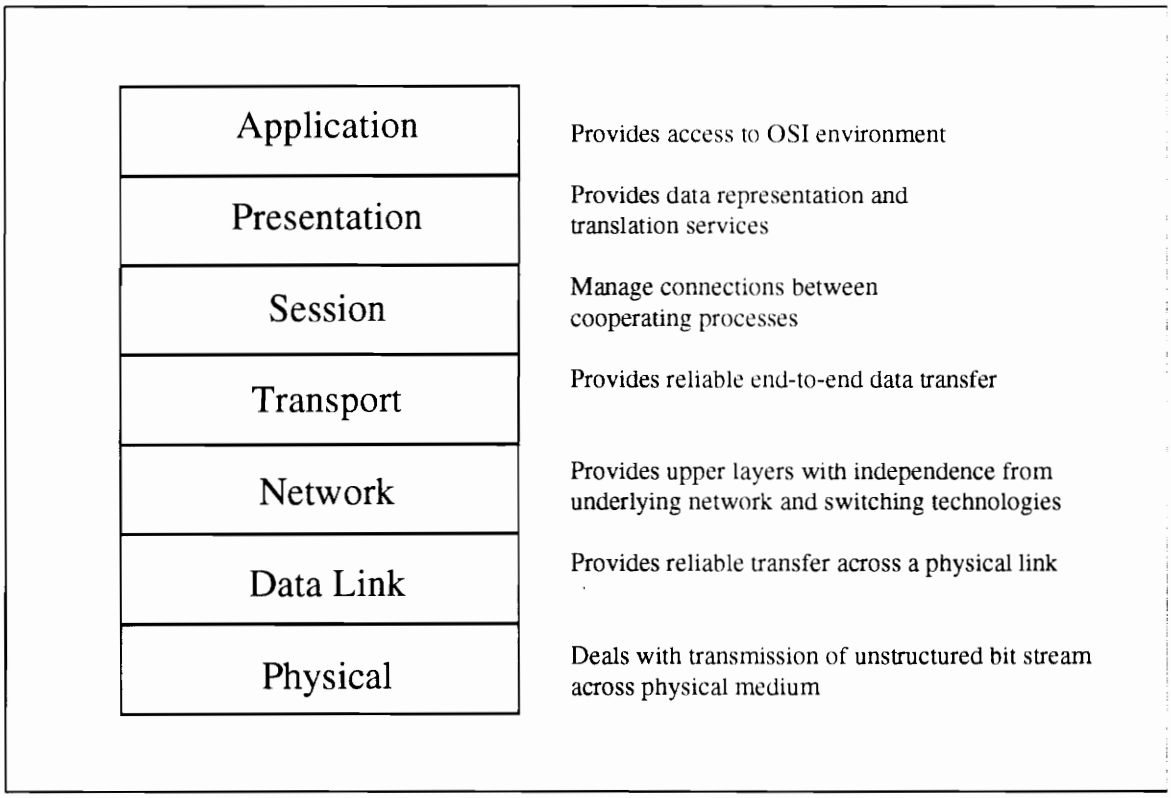


Figure 2.1 - Open Systems Interconnect model [Stall87].

2.1.2 Service Concepts

Central to the organization and description of many protocols is the idea of a *service*. A service describes a function provided by a given layer, such as connection establishment or data transfer. Each layer of the OSI reference model uses services

provided by the layer below it to provide a set of services to the layer above. The entity at layer N+1 that calls a layer N service is known as a *service user.* The entity that implements the services of layer N and makes them available to entities of layer N+1 is known as a *service provider.* A layer N service provider uses services provided by layer N-1 to provide enhanced services, known as the layer N services, to layer N+1 [Jain93].

The service provider makes services available to service users at the *Service Access Point* (SAP). From the service user's perspective, all services available are completely defined by the interface of the SAP [Rose90]. The SAP encapsulates the implementation of layer N services and may be changed internally as long as the external interface remains unchanged.

Communication protocol implementations can greatly benefit from an object-oriented approach for several reasons [Nicol93][Laven93]. First, object-oriented techniques can be helpful in managing the high degree of complexity involved in implementing communication protocols. Second, with the standards documents defining not only the services to be provided by a layer, but the interface to be presented, the standards themselves are somewhat object-oriented and translate well to object-oriented programming concepts.

2.1.3 Presentation Layer

Since this project deals with the representation and translation of data, it is related to the presentation layer. The major elements defined by the presentation layer include an abstract syntax for representing data and a transfer syntax for sharing data. Additionally, the negotiation at connection establishment of a presentation context, which defines the abstract syntax and the transfer syntax, is a service provided. This service is not addressed in this project since the goal was to develop a scheme that separated data representation from translation.

The abstract syntax provides a method for application entities to define data structures independent of the internal representation used for the data types involved. The abstract syntax definition of a data structure must be mapped to a concrete representation of that structure using a particular programming language on the machine being used. The transfer syntax is applied to the abstract syntax to obtain the representation of the data structure according to the canonical standard defined by the transfer syntax [Rose90]. Several approaches to providing an abstract syntax have been used. First, the approach used by Sun is to simply use the data definition facilities of an existing programming language, such as C. These data structures are then easily incorporated into application programs since the abstract representation of a data structure

maps directly to a concrete representation in the programming language. The second approach is to define a data definition language. This is the method used by the ISO in defining their data representation facilities. Data structures defined using such an abstract language must then be mapped to data structures of the programming language employed. In either case, the abstract syntax provides a way for the application entities, also known as presentation service users, to unambiguously define the data to be shared [Jain93].

An entity, within the presentation layer, that provides presentation layer services is known as a presentation service provider. The transfer syntax provides a method for presentation service providers to represent data in a standard format so it may be sent from one system to another regardless of differences in the internal representation of the data types used by the two systems. The transfer syntax defines the rules used to represent data in the standard format. A set of encoding rules defines the translation between the internal representation of the data types used and the standard representations defined by the transfer syntax. Figure 2.2 shows the relationship between presentation service users (application entities), presentation service providers, abstract syntax, and transfer syntax.

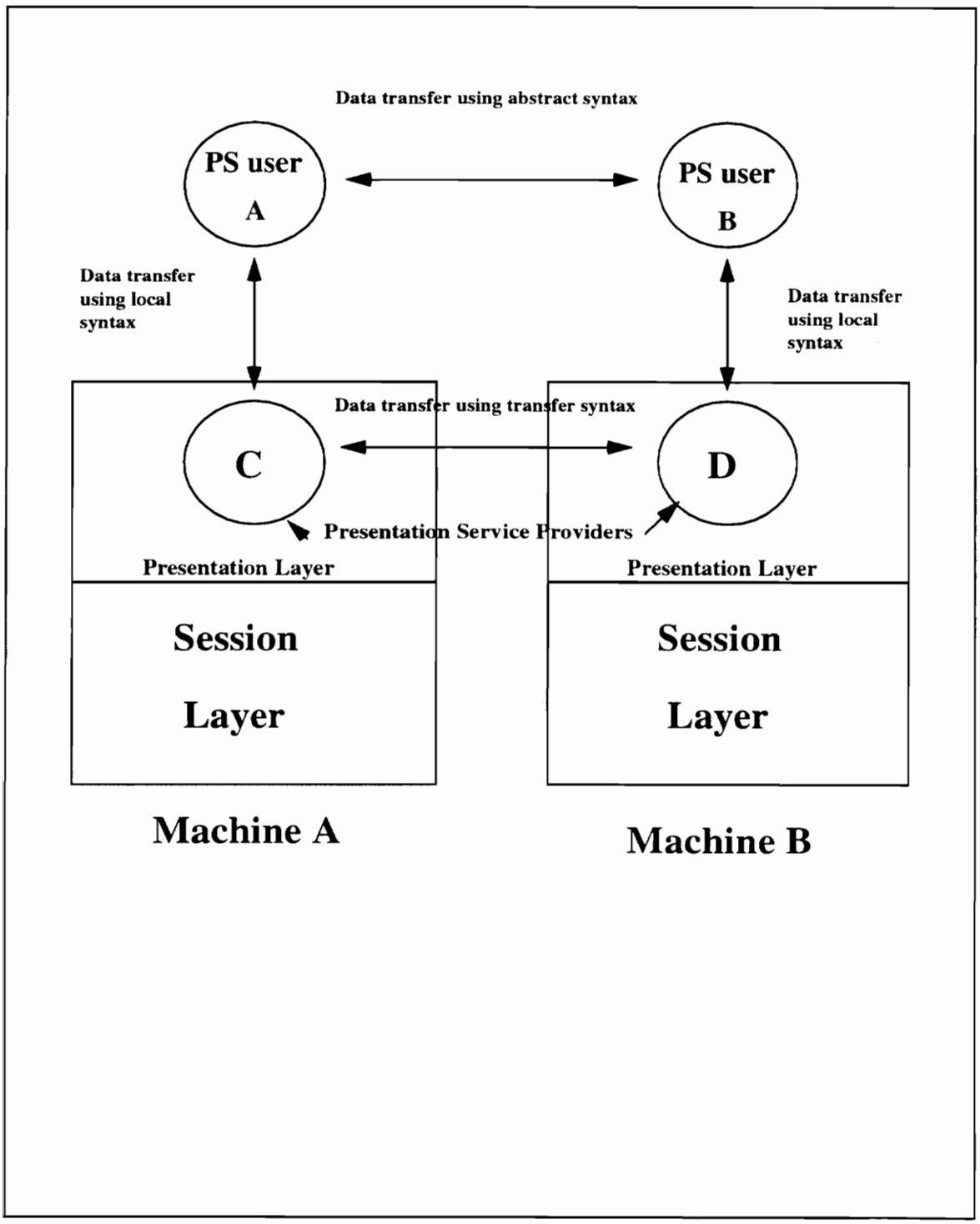


Figure 2.2 - Data Transfer Using Presentation Services

Presentation users A and B may share data by performing the following procedures. First, each presentation service user must have an abstract representation of the data to be shared. The presentation service users logically communicate using the abstract syntax as indicated by the top horizontal line in Figure 2.2. The presentation service users communicate the data to be shared with their presentation service providers using the local syntax defined by the particular machine and programming language being used. The presentation service providers then translate data between the local syntax (or internal representation) and the transfer syntax defined in the presentation context. Presentation service providers logically communicate using the transfer syntax as indicated by the bottom horizontal line in Figure 2.2. On the transmitting side, once translated to the standard representation, the data may be passed to the session layer for transmission to the remote system. On the receiving end of the connection, the session layer passes the data received to the presentation layer, where it is translated from the standard format to the local syntax of the receiving machine and passed to the presentation service user.

2.1.4 Overview of Class Library

Separating the representation of a shared data structure from the way it will be translated relies on the notion that data sharing consists of two parts, representation and translation. The class library is divided into two class hierarchies to reflect the idea of two parts of data sharing. There are also two levels of representation, abstract and concrete. Abstract representation is provided by an abstract notation that is independent of the internal representation of the data types on a particular machine. ASN.1 is an example of a language that allows abstract representation of data structures. The representation classes of the ASN class library also allow abstract representation of data structures since the class library may be used on any machine, regardless of the internal form used to store data. An abstract representation of a data structure must be mapped to a concrete representation of that data structure on each machine that will use it [Rose90]. For example, an ASN.1 INTEGER can be mapped to a C integer or long integer. This mapping can occur at compile time, as with the ISODE compiler that reads ASN.1 specifications and produces C language structures, or at language design time, as with the Sun RPC environment which uses C structures directly. The ASN representation classes fall into this second category since the concrete representation has already been chosen and is encapsulated in the classes themselves.

The translation portion of data sharing consists of applying a transfer syntax to the concrete representation of a data structure to obtain a standard representation of that data structure according the specified encoding rules. Each transfer syntax contains rules for encoding the basic data types such as integer, real, and string, and rules for combining the basic types to encode complex data structures, such as structures and unions. Previous attempts to provide representation and translation services have intermixed the translation code with the representation code, which ties a representation to a specific translation. To provide facilities that allow a representation to be translated by any transfer syntax, it is necessary to factor out the translation code and place it in a separate translation or *presentation element* class. The representation classes then invoke methods of the presentation element class to perform the translation. If the presentation element class can contain different encoding rules, the goal of providing flexible encoding services has been achieved.

In this project, the abstract and transfer syntaxes are embodied in two class hierarchies. The first provides data representation services (abstract syntax), while the second provides translation services (transfer syntax). The data representation services are provided through a set of classes that implement basic data types such as integer and real, and through several classes that provide the ability to combine data types to form more complex data structures. Since the combining is recursive, arbitrarily complex data structures can be built. The set of data types provided is based on the ASN.1 data

description language because it contains a rich set of types and most other description mechanisms are a subset of ASN.1. Thus, the data description class hierarchy is also called the ASN class hierarchy. Those working in a more restrictive environment, such as Sun's RPC, will only use a subset of the types available.

The data translation services are provided by two classes that implement the encoding rules of the BER and XDR standards and are structured in such a way that additional encoding rule sets may be added without modifications to any existing classes. These classes introduce the concept of a presentation element (PE) which encapsulates the encoding rules and the encoded form of a data structure. The PE abstraction is implemented by a C++ template that takes the transfer syntax to be used as a parameter. Since a PE will present the same interface regardless of the transfer syntax specified, the user may choose either the BER or XDR transfer syntax for encoding a data structure as long as the transfer syntax has rules for encoding all the data types involved. For example, the XDR encoding rules could not be chosen to encode a bit string since XDR is a byte-oriented syntax and is not able to represent or encode individual bit values.

When a presentation service user wishes to share data, the user first defines the data to be shared using the classes of the ASN class hierarchy. Then the user instantiates a PE with the transfer syntax to be used. When the data structure is assigned to the PE, the data structure is encoded using the rules specified in the PE declaration. Therefore,

the representation of shared data structures has been separated from the transfer syntax used to perform the sharing.

2.2 Abstract Data Types

The ASN class hierarchy provides a set of abstract data types that are used in building shared data structures. The ASN.1 language also provides abstract data types. However, the C names generated by an ASN.1 compiler are usually mangled to provide unique type names. This mangling is often identified as an impediment to application developers [Bran92]. The C++ ASN class hierarchy allows users to control the names of the user defined data types. The use of abstract data types of the ASN class hierarchy has several advantages. First, data structures are defined abstractly, independent of the implementation and internal representation of the class. For example, an abstract integer has the same meaning for an application regardless of the machine on which the application is running. However, the concrete representation of that abstract integer using a C language int can vary from machine to machine since some machines have 16-bit integers, others have 32-bit integers; some machines use a big endian representation, while others use little endian. The important point when defining the data that will be

shared, is that an integer will be sent, not how that integer will be represented internally on a particular machine. Second, users work directly with the predefined and user defined names rather than the awkward type names generated by a compiler.

The translation service is provided by the Presentation Element (PE), a C++ template class. The PE class provides the translation (encoding/decoding) routines for a specified transfer syntax. The translation from the internal format to the network representation may be viewed as a type conversion. When a data structure is to be encoded, it is type cast from it's internal representation to the network representation, much the same as an integer can be type cast to a floating point number. Decoding can be viewed similarly as a type cast from the network format to the internal format. These type conversions are accomplished by the assignment of an object with the internal form to an object that contains the network form, and vice versa. Therefore, a data structure is encoded for transmission on the network by assigning the data structure to a PE object which will then contain the encoded form of the data structure. Decoding is performed similarly by assigning the PE object that contains the encoded form of the data structure received from the network to an object defined with the appropriate data structure. For example, an integer with the value zero is encoded into a PE using the XDR rules by

```
INTEGER i(0);
PE<XDR> pe = i;
```

and a real is decoded from a PE using the BER rules by

```
PE<BER> pe;
// get data from network
REAL x = pe;
```

Any data structure, regardless of its complexity, can be encoded and decoded by the simple assignment statements shown above. The following section will show an example of encoding and decoding more complex data structures.

2.3 Interoperability Demonstration

This section presents an example of a client written with the classes of the data representation and translation hierarchies communicating with a server written using Sun's tool for writing applications using remote procedure calls, *rpcgen*. The application is the simple example presented in the *Sun Network Programming Manual* that prints a string on the console of a remote system. In addition to writing the remote print application, it is necessary to develop code to perform the remote procedure calls.

2.3.1 RPC Environment

A brief overview of the RPC environment will aid in understanding the example. Since applications using Sun RPC use TCP or UDP to communicate across the network, they must be bound to a port number that identifies the process sending or receiving the data on a particular machine. The client process must know the port number of the server in order to address the data properly. Certain servers operate on *well known ports* that are known across the network. For example, the File Transfer Protocol (FTP) server runs on port 21 on every system on the Internet. To avoid the requirement that all servers must be assigned well known ports, the *port mapper* was developed. When a server process starts, it allows the operating system to assign it a port number, and then registers that port number with the port mapper. When a client wishes to communicate with a server, it sends a query to the port mapper on the remote system, which runs on well known port 111. The port mapper returns the port number of the application specified in the query message. The client may then address data to that port number to communicate with the server.

As explained in Chapter 1, the RPC model of data sharing preserves the procedure call model of programming with which most programmers are familiar. To achieve this, the client process sends a message that contains the arguments necessary to perform the

work requested, and then waits for a response from the server. When the server receives a message, it performs its task and returns a message containing the response value. The client can be viewed as a caller in the procedure call model and the server can be viewed as a called subroutine.

2.3.2 Port Mapper Message

This section presents an example of querying the port mapper for the port number of a particular application. Since a port mapper query message is defined as an RPC message that contains a *mapping* structure as the user data, and the response is an RPC message with an integer as the user data, the first step in providing remote procedure call facilities is to define an RPC message and the mapping structure using the ASN class

```
class mapping : public SEQUENCE{
public:
        REQUIRED<INTEGER> prog;
        REQUIRED<INTEGER> vers;
        REQUIRED<INTEGER> prot;
        REQUIRED<INTEGER> port;

       void operator=(MetaPE& pe) { SEQUENCE::operator=(pe); }
        mapping() : prog(this), vers(this), prot(this), port(this){}
};
```

Figure 2.3 - Mapping Structure

library. Figure 2.3 shows the definition of the mapping structure and the classes required for a call message are shown in Figure 2.4. A user's guide for building shared data structures is given in Appendix A and a complete definition of an RPC message is given in Appendix B. [Sun88] presents the RPC and port mapper protocols.

The definition of the mapping data structure show in Figure 2.3 introduces several new classes of the ASN class hierarchy. Since the mapping data structure is defined as a C structure [Sun88], it is defined as a C++ class that inherits from the *SEQUENCE* class of the ASN class library. Additionally, since the ASN.1 language allows elements to be optional, it is necessary to specify if the members of the sequence are required or optional. The C++ templates *REQUIRED* and *OPTIONAL* perform this function. The mapping class must keep a list of its member elements so the individual elements may be encoded when the mapping encoding is performed. To accomplish this, it is necessary to call the REQUIRED and OPTIONAL element's constructor passing a pointer to the data structure being defined. Such a pointer is provided by C++, called the *this* pointer. Finally, an assignment operator is defined since assignment operators are not inherited in C++.

Figure 2.4 presents several new types that are used to define a remote procedure call message. The *rpc_msg* type is a C++ template, where the template argument specifies the type of the user data in the message. An RPC message is a sequence with

```
template <class T> class call_body : public SEQUENCE {
public:
        REQUIRED<INTEGER>     rpcvers;
        REQUIRED<INTEGER>     prog;
        REQUIRED<INTEGER>     vers;
        REQUIRED<INTEGER>     proc;
        REQUIRED<my_auth>     cred;
        REQUIRED<my_auth>     verf;
        REQUIRED<T >          Args;     // procedure specific


       void operator=(MetaPE& pe) { SEQUENCE::operator=(pe); }
       void operator=(call_body& v) {}
       call_body() : rpcvers(this), prog(this), vers(this), proc(this),
                          cred(this), verf(this), Args(this) {}
};

template <class T> class body_type : public DISC_UNION {
public:
    ChoiceElement<call_body < T >, CALL>   cbody;
    ChoiceElement<reply_body< T >, REPLY>  rbody;

    void operator= (MetaPE& pe) { DISC_UNION::operator=(pe); }
    void operator= (body_type& v) {}
    body_type() : cbody(this), rbody(this) {};
};

template <class T> class rpc_msg : public SEQUENCE {
public:
    REQUIRED<INTEGER>          xid;
    REQUIRED<body_type< T > >  body;

    void operator=(MetaPE& pe) { SEQUENCE::operator=(pe); }
    rpc_msg() : xid(this), body(this) {}
};
```

Figure 2.4 - RPC/Port Mapper Message

two required elements, *xid* of type INTEGER, and *body* of type body_type. The definition of the body_type class introduces a new class *DISC_UNION*. The DISC_UNION class implements the discriminated union as defined by the XDR standard. This is similar to the *CHOICE* class that implements the CHOICE type of the ASN.1 language. The difference between DISC_UNION and CHOICE lies in the encoding. In both the DISC_UNION and CHOICE encodings, it is necessary to include type information for the type being encoded to allow proper decoding. For the Basic Encoding Rules, this is not a problem since type information is always encoded with every data type. XDR on the other hand never encodes type information. To inform the decoder of the type of the data in the union, XDR encodes an integer, the discriminant, before the data contained in the union. For example, an encoding of a body_type instance consists of an integer discriminent followed by a call_body, if the discriminent is equal to CALL, or followed by a reply_body if the discriminent is equal to REPLY. The ChoiceElement template performs a function similar to the REQUIRED and OPTIONAL templates used in the construction of sequences. The ChoiceElement template allows the enclosing class to maintain a list of the members of the union, and it defines an assignment operator for each member that sets the type information of the enclosing class. This type information is used to perform a proper encoding of the information stored in the DISC_UNION or CHOICE.

Figure 2.5 shows a C++ function that takes a remote host name, the program number of a remote program, and the version number, and returns the port number that the program is using on the remote system. Since this function contains the details required for setting up a port mapper message, it would ideally be defined as a member function of a base class so users would not need to be concerned about the details associated with querying the port mapper. Not shown in Figure 2.5 is the *udp_socket* class that contains the code that creates a socket and sends and receives UDP packets on the network. This code is quite simple and is not necessary to understand the example shown, but is presented in Appendix B.

```
int get_port (char* rhost, INTEGER prog, INTEGER vers)
{
        rpc_msg<mapping>        msg;   // the query message
        rpc_msg<INTEGER>        reply; // the response message
        mapping                 map;

        map.prog = prog;
        map.vers = vers;
        map.prot = 17;                  // hardcoded to UDP for this example
        map.port = 0;                   // not used in query message

        msg.xid = 1;
        msg.body.type               = 0;
        msg.body.cbody.rpcvers      = 2;
        msg.body.cbody.prog         = 100000;    //program number of port mapper
        msg.body.cbody.vers         = 2;
        msg.body.cbody.proc         = 3;         // procedure number of Get Port
        msg.body.cbody.cred.a       = 0;         // unused
        msg.body.cbody.cred.b       = 0;         // unused
        msg.body.cbody.verf.a       = 0;         // unused
        msg.body.cbody.verf.b       = 0;         // unused
        msg.body.cbody.Args         = map;

        PE<XDR>     pe(msg);                     // encode the message

        udp_socket     remote;
        remote.a_request ("udp", rhost, 111);   // socket initialization
        remote.Write (pe.base, pe.data - pe.base);
        remote.Read  (pe.base, 132);
        pe.data = pe.base;

        reply   = pe;                           // decode the response
        // do some error checking

        return reply.body.rbody.areply.reply_data.results;
}
```

Figure 2.5 - Port Mapper Query

2.3.3 Complete Remote String Print Application

To implement the remote string print operation, a simple remote operation class is defined in Figure 2.6. The parameter to the template specifies the type of the user data to be passed to the remote system. After the port mapper is queried for the remote applications port, an RPC message with the type parameter is built and sent to the server on the remote system. The server decodes the message, performs its task, and returns a response to the client. In this simple example, the return value is simply an integer indicating the completion status of the print operation.

One can easily imagine a more general remote operation template that specifies the user data being sent and received. Figure 2.7 shows how this simple remote operation class can be used to implement the example from Chapter 2 of the *Sun Network Programming Manual*, shown in Figure 2.8.

```
class ROP {
public:
        INTEGER prog, vers, proc;
        ROP (INTEGER prg, INTEGER v, INTEGER prc) :
                        prog(prg), vers(v), proc(prc) {}

        int call_rop (char* rhost, OCTET_STRING s)
        {
                rpc_msg<OCTET_STRING> msg;
                rpc_msg<INTEGER>      reply;
                int port = get_port(rhost, prog, vers);

                msg.xid = 1;
                msg.body.type                = 0;
                msg.body.cbody.rpcvers       = 2;
                msg.body.cbody.prog          = prog;
                msg.body.cbody.vers          = vers;
                msg.body.cbody.proc          = proc;
                msg.body.cbody.cred.a        = 0;          // unused
                msg.body.cbody.cred.b        = 0;          // unused
                msg.body.cbody.verf.a        = 0;          // unused
                msg.body.cbody.verf.b        = 0;          // unused
                msg.body.cbody.Args          = s;

                PE<XDR> pe(msg);                           // encode the message
                udp_socket  remote;
                remote.a_request ("udp", rhost, port);
                remote.Write (pe.base, pe.data - pe.base);
                remote.Read  (pe.base, 132);
                pe.data = pe.base;

                reply = pe;                                // decode the response
                // do some error checking

                return reply.body.rbody.areply.reply_data.results;
        }
};
```

Figure 2.6 - Remote Operation Class

```
main(int argc, char** argv)
{
        INTEGER prog = 99;
        INTEGER vers = 1;
        INTEGER proc = 1;

        if (argc != 3) {
                cout << "usage:  " << argv[0] << " host message\n";
                exit(1);
        }

        OCTET_STRING message = argv[2];
        ROP     rop(prog, vers, proc);
        int stat = rop.call_rop (argv[1], message);

        if (!stat)
                cout << "couldn't print your message\n";
        else
                cout << "Message Delivered!\n";
}
```

Figure 2.7 - Remote String Print

```
program MESSAGEPROG {
        version         MESSAGEVERS {
                int     PRINTMESSAGE(string) = 1;
        } = 1;
} = 99;
```

Figure 2.8 - msg.x [Sun88]

The file *msg.x* shown in Figure 2.8 is compiled using the *rpcgen* compiler along with the files *msg.h,* which contains preprocessor constants for MESSAGEPROG, MESSAGEVERS, and PRINTMESSAGE, *msg_clnt.c,* which contains the client *stub* routines, and *msg_svc*, which contains the server *stub* routines. The stub routines contain code to marshall the arguments for the remote procedure, to perform data translation, and to call the necessary network routines. Figure 2.9 shows the client code that sets up a network connection and calls the stub routine that invokes the remote procedure.

Comparing the C++ implementation shown in Figure 2.7 and the Sun RPC implementation shown in Figure 2.9 shows that the C++ remote string print application using the ASN class hierarchy is as simple as the Sun RPC version, but does not require the rpcgen compiler.

```
#include <stdio.h>
#include <rpc/rpc.h>          /* always needed */
#include "msg.h"

main(argc, argv)
        int     argc;
        char *argv[];
{
        CLIENT          *cl;
        int             *result;
        char            *server;
        char            *message;

        /* argument checking omitted here */

        server          = argv[1];
        message         = argv[2];
        cl = clnt_create (server, MESSAGEPROG, MESSAGEVERS, "tcp");
        if (cl == NULL) {    /* error */
                clnt_pcreateerror(server);
                exit(1);
        }

        result = printmessage_1 (&message, cl);    /* call stub that invokes
                                                      remote proc*/
        if  (*result == 0) {    /* error */
                fprintf (stderr, "%s: %s couldn't print your message\n",
                        argv[0], server);
                exit(1);
        }

        printf ("Message delivered to %s!\n", server);
        exit(0);
}
```

Figure 2.9 - Remote Print Message [Sun88]

3.0 Implementation

Chapter 2 discussed the use of the ASN class library for building shared data structures and the use of the presentation element class to apply a transfer syntax to the concrete representation of a shared data structure. Chapter 3 concentrates on the organization of these classes into two separate, but related, class hierarchies. The organization permits flexible encodings and allows the user to directly control the names of the types and identifiers. The organization of the representation class hierarchy is presented in detail, followed by the structure of the translation class hierarchy.

3.1 Representation Class Hierarchy

Figure 3.1 shows the organization of the ASN data representation class hierarchy. An abstract data type, *AsnType*, is used as a base class for all the representation classes in order to express the characteristics common to all data structures and to provide a way to reference a representation object without specific information about the type of the

object. All representation classes must inherit from AsnType and implement the pure virtual functions declared in AsnType. The classes *AsnPrim* and *AsnCons* are subclasses of AsnType and used for representing simple and constructed types, respectively. *SEQUENCE, SEQUENCE_OF, SET,* and *SET_OF* are subclasses of AsnCons, representing more specific constructed types.

Figure 3.2 shows the definition of the class AsnType, which defines the external interface presented by all representation classes. The class AsnType contains virtual functions for an assignment operator, the *tag* method that returns type information, an encode and decode method that provide the link to the translation class hierarchy, the *required, optional, and default* methods that are used to determine the status of elements of a constructed type during decoding, and the *print* and *indent* methods used for printing the data structures. Some members, such as encode and decode, are pure virtual functions forcing each subclass of AsnType to have an implementation of these functions in order to instantiate an object of the class. Other members, such as primitive and constructed, are virtual, but provide a default implementation that may be overridden by a subclass. For example, AsnCons will implement a *constructed* member that hides the constructed method defined in AsnType and returns true to indicate this is a constructed type.

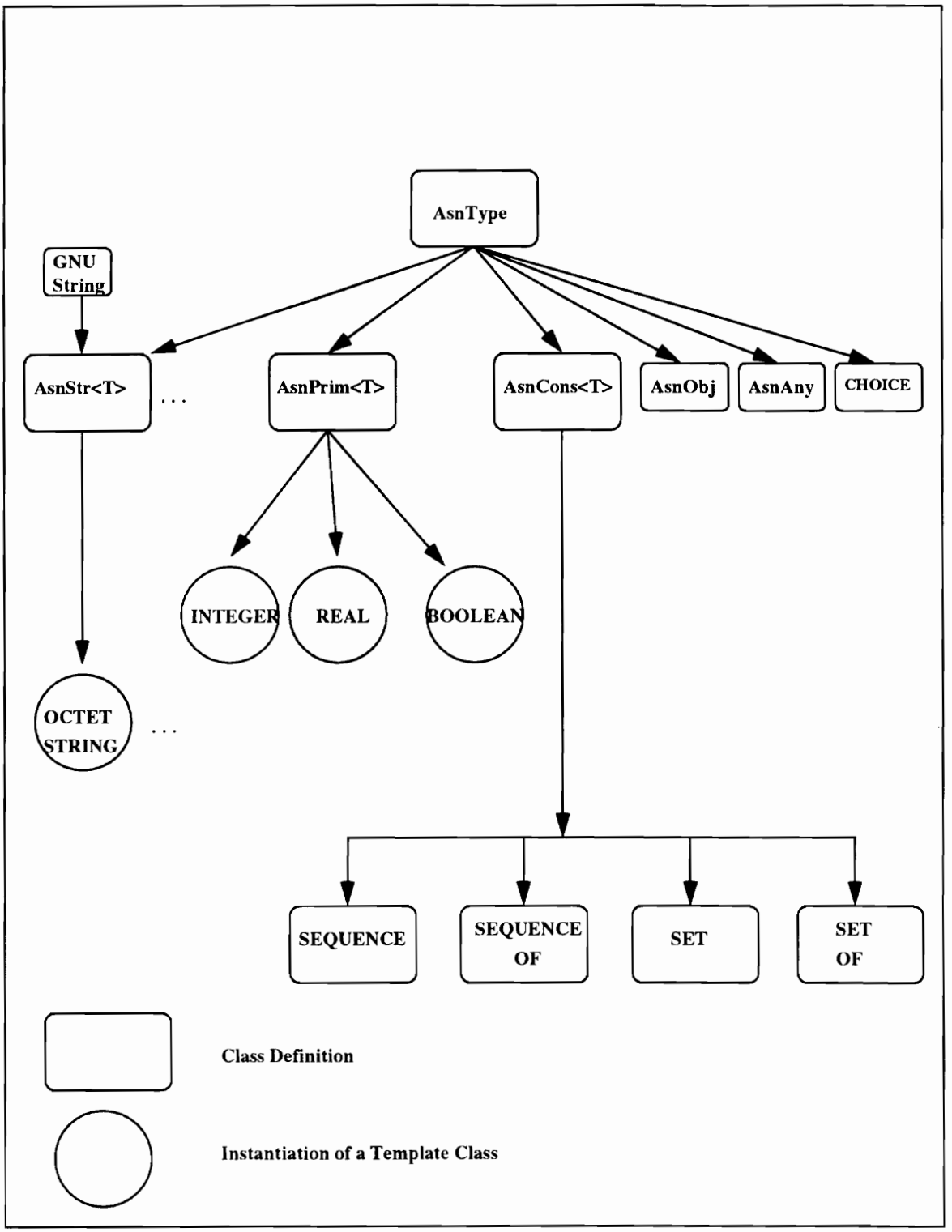


Figure 3.1 - ASN Class Hierarchy

```
class AsnType {
protected:
        void indent (int tab)           { for int i = 0; i < tab; i++) cout << "      "; }
public:

        enum { PRIM_FORM, CONS_FORM };

        virtual void operator= (const AsnType& ) {}

        virtual int encode (MetaPE* pe) = 0;
        virtual int decode (MetaPE* pe) = 0;

        virtual const AsnTag& tag() = 0;

        static int primitive()          { return 0; }
        static int constructed()        { return 0; }
        static int form()               { return primitive() ? PRIM_FORM :
                                                               CONS_FORM;}

        virtual int required () const   { return 0; }
        virtual int optional () const   { return 0; }
        virtual int defaults () const   { return 0; }

        virtual void print ( int tab = 0) = 0;

        // for building constructed types
        virtual int add (AsnType* ) { return -1; }
};
```

Figure 3.2 - AsnType

The class AsnTag is used to encapsulate type information. The class contains three data members to store the class, form, and identifier associated with the tag. These three components come from the ASN.1 description of a tag. The only methods of the AsnTag class are a constructor and a print method. The definition of the AsnTag class is shown in Figure 3.3.

```
typedef octet                   TagClass;
typedef octet                   TagForm;
typedef unsigned short          TagId;

const TagClass UNIVERSAL        = 0x0;
const TagClass APPLICATION      = 0x1;
const TagClass CONTEXT          = 0x2;
const TagClass PRIVATE          = 0x3;

struct AsnTag {

    TagClass   cl;      // class
    TagForm    fm;      // form
    TagId      id;      // identifier

    AsnTag(TagClass c, TagForm f, TagId i) : cl(c), fm(f), id(i) {}
    void print();       // implementation not shown
};
```

Figure 3.3 - AsnTag

3.1.1 Primitive Types

The primitive data types consist of numeric types (integer, and real), a logical type (boolean), and various string types (octet string, etc.) as shown in Figure 3.1. All primitive data types have similar members that perform type casting and assignment, and apply the transfer syntax. The only difference among the primitive data types is the types on which the members operate. The C++ template class AsnPrim reflects this similarity. Thus, the type AsnPrim<int> can store integer values and represent the ASN.1 INTEGER type, while AsnPrim<double> can store real number values and represent the ASN.1 REAL type. AsnPrim is used to define the simple primitive types, integer, real, and boolean. The string primitive types are defined with their own classes to provide operators that copy the string values rather than assigning one pointer to another.

Figure 3.4 shows the definition of the AsnPrim class. Notice that the pure virtual functions of AsnType are implemented in AsnPrim and the primitive member of AsnPrim hides the primitive member of AsnType so that primitive will return true for the AsnPrim class. The additional arguments to the template, *C* and *I*, contain type information that is used by some encoding rules such as BER. This information is stored in an object of class *AsnTag*, which is used for storing, printing, encoding and decoding type information.

Several details not important to the discussion, such as the implementation of the print method, are omitted.

With the AsnPrim template fully defined, the primitive data types for data sharing can be defined with simple type definition statements as shown in Figure 3.5. In addition to being simple, this method for defining types is extensible since new types, such as unsigned integer, can be defined by a single type definition statement.

```
template<class T, TagClass C, TagId I> class AsnPrim : public AsnType {
protected:
    T   prim;  /* primitive value */

public:

    int primitive() const       { return 1; }

    const  AsnTag& tag()     {static AsnTag t(C, PRIM_FORM, I); return t; }
    static AsnTag& ChoiceTag(){ static AsnTag t(C, PRIM_FORM, I); return t; }

    inline AsnPrim()                  {}
    inline AsnPrim(AsnType* asn)   { asn->add(this); }
    inline AsnPrim(const T& v)        { prim = v; }
    inline AsnPrim(const AsnPrim& p) { prim = p.prim; }
    inline AsnPrim(MetaPE& pe)     { pe.start(); pe.pe2prim(prim, tag()); }

    void operator=(const T& v)              { prim = v; }
    void operator=(const AsnPrim& p)        { prim = p.prim; }
    void operator=(const AsnType& p)        { typedef AsnPrim& mytype;
                                              prim = ((mytype)p).prim;}
    inline void operator=(MetaPE& pe)       { pe.start(); pe.pe2prim(prim, tag());}
    inline operator T() const               { return prim; }

    int encode(MetaPE* pe)                  { return pe->prim2pe(prim, tag()); }
    int decode(MetaPE* pe)                  { return pe->pe2prim(prim, tag()); }
    void print (int tab = 0);
};
```

Figure 3.4 - AsnPrim

```
typedef      int              AsnBool;
typedef      long             AsnInt;
typedef      double           AsnReal;

typedef AsnPrim<AsnBool,  UNIVERSAL, 1>    BOOLEAN;
typedef AsnPrim<AsnInt,   UNIVERSAL, 2>    INTEGER;
typedef AsnPrim<AsnReal,  UNIVERSAL, 9>    REAL;
typedef AsnPrim<AsnInt,   UNIVERSAL, 10>   ENUMERATED;
```

Figure 3.5 - Primitive Type Definitions

The character string classes, while also considered primitive by the ASN.1 language definition, are actually arrays of characters. The C and C++ languages do not implement these directly, but provide library functions to manipulate arrays of characters through a pointer to the first element of the array. Since the operators available for strings are semantically different from those available for integer, real, and boolean, the character string types are not implemented with the AsnPrim template. For this reason and to provide a more user friendly interface, it was decided to implement the character string classes using the GNU String class [Lea92]. This is accomplished by defining a class, *AsnStr*, similar to AsnPrim, but that inherits from both AsnType and String. AsnStr is also a template, but does not require the type T parameter since all instances of the class will be represented internally with a GNU String. AsnStr implements the same functions as AsnPrim, but calls members of the parent class String to manipulate the string values. The definition of AsnStr is shown in Figure 3.6. Each string type is then defined with a type definition statement, show in Figure 3.7.

```
template<TagClass C, TagId I> class AsnStr : public String, public AsnType {
typedef AsnStr& mytype;
public:
        int primitive() const { return 1; }

        const  AsnTag& tag()  { static AsnTag t(C, PRIM_FORM, I); return t; }

        inline AsnStr() : String()                      {}
        inline AsnStr(AsnType* asn)                     { asn->add(this); }
        inline AsnStr(const String& x) : String(x)  {}
        inline AsnStr(const SubString&  x) : String(x)      {}
        inline AsnStr(const char* t) : String(t)        {}
        inline AsnStr(const char* t, int len) : String(t, len) {}
        inline AsnStr(char c) : String(c)               {}
        inline AsnStr(MetaPE& pe)           { pe.start(); pe.pe2prim(*this, tag()); }

        inline void operator=(const AsnType& a)
                                        {String::operator=((mytype)a);}
        inline void operator=(const String& y)      { String::operator=(y); }
        inline void operator=(const char* y)        { String::operator=(y); }
        inline void operator=(char c)               { String::operator=(c); }
        inline void operator=(const SubString&  y) { String::operator=(y); }
        inline void operator=(MetaPE& pe) { pe.start(); pe.pe2prim(*this, tag()); }

        int encode(MetaPE* pe) { return pe->prim2pe(*this, tag()); }
        int decode(MetaPE* pe) { return pe->pe2prim(*this, tag()); }
        void print (int tab = 0);
};
```

Figure 3.6 - AsnStr

```
typedef AsnStr<UNIVERSAL,                    4>
OCTET_STRING;

typedef IMPLICIT<GraphicString, UNIVERSAL,   7>
ObjectDescriptor;
typedef IMPLICIT<IA5String, UNIVERSAL,       18>  NumericString;
typedef IMPLICIT<IA5String, UNIVERSAL,       19>  PrintableString;
typedef IMPLICIT<OCTET_STRING, UNIVERSAL,    20>  T61String;
typedef T61String                                 TeletexString;
typedef IMPLICIT<OCTET_STRING, UNIVERSAL,    21>  VideotexString;
typedef IMPLICIT<OCTET_STRING, UNIVERSAL,    22>  IA5String;
typedef IMPLICIT<VisibleString, UNIVERSAL,   23>  UTCTime;
typedef UTCTime                                   UniversalTime;
typedef IMPLICIT<VisibleString, UNIVERSAL,   24>
GeneralizedTime;
typedef GeneralizedTime
GeneralisedTime;
typedef IMPLICIT<OCTET_STRING, UNIVERSAL,    25>  GraphicString;
typedef IMPLICIT<OCTET_STRING, UNIVERSAL,    26>  VisibleString;
typedef VisibleString                             ISO646String;
typedef IMPLICIT<OCTET_STRING, UNIVERSAL,    27>  GeneralString;
typedef IMPLICIT<OCTET_STRING, UNIVERSAL,    27>  CharacterString;
```

Figure 3.7 - Character String Type Definitions

The *AsnBits* class is implemented exactly the same as AsnStr, except that it inherits from AsnType and the GNU class BitString. The members of AsnBits call functions in the parent class BitString to manipulate the string of bits.

One additional primitive type is defined in the ASN.1 standard that has no corresponding type in the C language. This OSI primitive type is referred to as an object identifier. An object identifier is used to represent a node in the object identifier tree. Each node in the tree represents a registered information object such as a standards document, a module specification, or a remote operation. An object identifier is represented by a list of integers that specify which branch should be taken in traversing the tree, starting from the root, to arrive at the specified node. The most common form for specifying the value of an object identifier is a dot or space separated list of integers. While the semantics of an object identifier are quite different from a string, it is quite natural to implement the OBJECT_IDENTIFIER class by inheriting from AsnType and the GNU String class. The GNU String operators are then available for manipulating the object identifier. OBJECT_IDENTIFIER must be distinct from the string classes since the encoding of an object identifier is different from the encodings of any other types. For example, an OBJECT_IDENTIFIER administered jointly by the ISO and the International Telephone and Telegraph Consultative Committee (ISO/CCITT) telephone number attribute can be defined by

```
OBJECT_IDENTIFIER       phone_obj_id = "2.5.4.20";
```

Refer to Appendix B for the details of the implementation of the OBJECT_IDENTIFIER class.

3.1.2 Constructed Types

As shown in Figure 3.1, all constructed types inherit from the class AsnCons. The AsnCons class implements those pure virtual functions of AsnType which are the same for all constructed types. For example, all constructed types will require an encode method that encodes each member of the data structure, so the encode method is implemented in the AsnCons class. Decoding a constructed type is different, however, since the SEQUENCE and SEQUENCE_OF maintain the order of their elements, while SET and SET_OF are not required to do so. Therefore, the decode method must be implemented in the subclasses of AsnCons. The storage of members of a constructed type is via a list of pointers to the individual members. AsnCons is defined as a C++ template with three arguments, the type of the members, and the class and identifier used to build the tag. Figure 3.8 shows the definition of AsnCons.

```
template<class T, TagClass C, int I> class AsnCons : public AsnType {
protected:
        T**     cons;
        int     size;
        int     n;
        typedef AsnCons& mytype;
public:
        AsnCons(int s = 16) : n(-1)             { cons = new T*[size = s]; }
        AsnCons(AsnType* asn, int s = 16) : n(-1){
                        cons = new T*[size = s]; asn->add(this); }
        AsnCons(MetaPE& pe)                     { pe.start(); (void) decode(&pe); }
        ~AsnCons()                              { delete cons; }

        void operator=(const AsnCons& v) {
                for (int i = 0; i <= n; i++)
                        *cons[i] = *(v.cons[i]);
        }
        void operator=(const AsnType& a)  { operator=((mytype)a); }
        void operator=(MetaPE& pe)        { pe.start(); (void) decode(&pe); }

        constructed() const               { return TRUE; }

        const AsnTag& tag() { static AsnTag t(C, CONS_FORM, I); return t; }
        static const AsnTag& ChoiceTag() {
                static AsnTag t(C, CONS_FORM, I); return t; }

        int add(T* asn)             { cons[++n] = asn; return n + 1; }

        int encode(MetaPE* pe) {
                int result = pe->cons2pe(tag(), n + 1);
                for (int i = 0; i <= n && result == 0; i++)
                        result = cons[i]->encode(pe);
                pe->end();
                return result;
        }

        void print(int tab = 0);
};
```

Figure 3.8 - AsnCons

All constructed types maintain a list of pointers to the individual data members of the class, *cons*. The cons array is allocated by the constructor, but contains no values until the *add* member is called. Each member of the constructed data structure must call the add member with its *this* pointer to be added to the cons list. When a constructed type, such as a sequence, is constructed as shown in Figure 3.9, the base class constructors are called first, followed by the constructors for each member. Note the importance of abstraction. Any AsnType can be in the list and manipulated appropriately.

```
class my_seq : public SEQUENCE {
public:
        REQUIRED<INTEGER> i;
        REQUIRED<INTEGER> j;

        my_seq() : i(this), j(this) {}
};
```

Figure 3.9 - Constructed Type

The my_seq constructor invokes the SEQUENCE constructor which invokes the AsnCons constructor which calls the AsnType constructor. Following this, the REQUIRED constructor for i is called, which calls the AsnPrim constructor with a pointer to an instance of my_seq, followed by the AsnType constructor. Then the body of the AsnPrim constructor is executed which invokes the add method of my_seq through the pointer that was passed to the constructor, passing a pointer to i. The add method appends the pointer to i to the cons array. The same steps are then executed for j. After all constructors are called, the cons array in my_seq contains pointers to i and j. The same sequence of

events occurs with the OPTIONAL and DEFAULT templates. The REQUIRED, OPTIONAL, and DEFAULT templates are shown in Figure 3.10.

```
template<class T> struct AsnReqElem : public T {

   virtual void operator=(const T& v)      { T::operator=(v); }
   int                required() const     { return 1; }
};


template<class T> struct AsnOptElem : public T {

   inline void operator=(T& v)             { T::operator=(v); }
   int              optional() const       { return 1; }
};


template<class T> struct AsnDefElem : public T {
public:
   T def_val;

   inline void  operator=(T& v)            { T::operator=(v); }
   T                default_value()        { return def_val; }
   int              defaults() const       { return 1; }
};

#define REQUIRED  AsnReqElem
#define OPTIONAL  AsnOptElem
#define DEFAULT   AsnDefElem
```

Figure 3.10 - REQUIRED, OPTIONAL, and DEFAULT Templates

Encoding any of the constructed types can be performed by the same algorithm: call an initialization routine that encodes any type information required by the transfer syntax and then call the encoder for each member of the data structure. The encode routine in AsnCons calls *cons2pe* to perform any constructed type specific encoding and then loops through the cons array, calling the encoder for each member. Notice that this preserves the order of the elements in the encoding as required by the SEQUENCE and SEQUENCE_OF. This same encode routine can be used to encode an unordered data structure, such as a SET, since it is not incorrect to maintain the order of the elements in the encoded form of the data structure. However, the same decode routine cannot be used since it would be incorrect to assume the elements of an encoded form of a SET are in the same order they were added to the cons array. Therefore, it is necessary to define subclasses of AsnCons for the four constructed types as shown in Figure 3.1. Figure 3.11 shows the definition of the SEQUENCE class. Notice that other than the constructors and assignment operator, which are not inherited in C++, decode is the only method that is implemented in the SEQUENCE class. The other constructed types are similar to the SEQUENCE class and are not presented here since they provide no further insight. Refer to Appendix B for the complete definition of these classes.

```
class SEQUENCE : public CONSTRUCTED<AsnType, UNIVERSAL, 16> {
public:
    SEQUENCE(int sz = 16) :
       CONSTRUCTED<AsnType, UNIVERSAL, 16> (sz){}
    SEQUENCE(AsnType* asn) : CONSTRUCTED<AsnType,
       UNIVERSAL,16> (asn) {}
    void operator= (MetaPE& pe) {
       CONSTRUCTED<AsnType, UNIVERSAL, 16>::operator= (pe);
    }
    int decode(MetaPE* pe)
    {
        int result;
        int i = 0;

        if (pe->cmptag(tag()) == TRUE)
            result = pe->pe2cons(tag());
        else
            return -1;

        while (i <= n && result == 0) {

            register AsnType* asn = cons[i++];

            if ((result = pe->cmptag(asn->tag())) == TRUE)
                result = asn->decode(pe);
            else if (asn->optional())
                continue;
            else if (asn->defaults())
                asn->set_default();
            else
                return -1;
        }

        return result;
    }
};
```

Figure 3.11 - SEQUENCE

3.1.3 Choice

The CHOICE class allows construction of a data structure that takes exactly one value from among one or more distinct types, similar to a union in many programming languages. CHOICE maintains a discriminant that determines the type of the value currently stored. The CHOICE class inherits from AsnType and, similar to AsnCons, it maintains a pointer to one object of each type from which it may take values. These pointers are stored in an array *cons* and are valued by the add member during the construction process in the same manner constructed types are built. Each member of the CHOICE must be a *ChoiceElement*, which is a template whose first parameter is the type and whose second parameter is the value associated with the discriminent when the CHOICE contains a value of the specified type. Thus, the definition of a CHOICE is similar to the definition of a SEQUENCE, with one member for each type the CHOICE may hold, and with the REQUIRED, OPTIONAL, or DEFAULT template replaced with ChoiceElement. Figure 3.12 shows the definition of the CHOICE class, followed by the MetaChoiceElement and ChoiceElement in Figure 3.13.

The MetaChoiceElement class serves as a base class for the ChoiceElement template. The CHOICE class may then refer to any ChoiceElement, regardless of its type parameter, as a MetaChoiceElement, since all instances of the ChoiceElement template are more specific types of the MetaChoiceElement class.

```
class CHOICE : public AsnType {
public:

    int type;
    int size;
    int nelem;
    MetaChoiceElem** cons;
    typedef MetaChoiceElem* ASNP;

    CHOICE(int s = 16) : type(-1),nelem(0) { cons = new ASNP[size = s]; }

    void operator=(CHOICE& c)     {
        if (type >= 0)
            cons[type]->destroy();
        type = c.type;
        cons[type]->create();
        *(cons[type]) = *(c.cons[type]);
    }
    void operator=(MetaPE& pe)     { decode (&pe); }
    int add(MetaChoiceElem* asn)   { cons[nelem++] = asn; return nelem; }
    const AsnTag& tag()            { return cons[type]->tag(); }
    void set_type(int n)           { type = n; }
    void destroy()                 { cons[type]->destroy(); }
    void print(int tab = 0)        { cons[type]->print(tab); }
    int encode(MetaPE* pe)         { return cons[type]->encode(pe); }
    int decode(MetaPE* pe) {
        for (type = 0; type < nelem; type++)
            if (pe->cmptag(tag()) )
                break;
        return cons[type]->decode(pe);
    }
};
```

Figure 3.12 - CHOICE

```
class MetaChoiceElem : public AsnType {
public:
    virtual AsnType* get_value()    = 0;
    virtual void     create()       = 0;
    virtual void     destroy()      = 0;
    virtual void     operator       =(AsnType&) {}
};

template <class T, int n> class ChoiceElement : public MetaChoiceElem {
typedef ChoiceElement& mytype;
public:
    T*      value;
    CHOICE* m;

    ChoiceElement (CHOICE* asn)          { m = asn; asn->add(this); }
    const AsnTag& tag()                  { return T::ChoiceTag(); }
    int encode(MetaPE* pe)               { return value->encode(pe); }
    int decode(MetaPE* pe)               { if (!value)
                                               value = new T;
                                           return value->decode(pe); }
    void print (int tab = 0)             { value->print(tab); }
    void create ()                       { value = new T; }
    void destroy()                       { delete value; }
    operator T&()                        { return *value; }
    AsnType* get_value()                 { return value; }
    void operator= (T& v)                { m->set_type(n);
                                           if (!value)
                                               create();
                                           value->operator=(v); }
    void operator=(AsnType& asn)         { m->set_type(n);
                                           value= new T;
                                           mytype cep = (mytype)asn;
                                           *value = *(cep.value); }
    void operator=(ChoiceElement& ce)    { m->set_type(n);
                                           value = new T;
                                           *value = *(ce.value); }
};
```

Figure 3.13 - MetaChoiceElement and ChoiceElement

Figure 3.14 shows the definition of a choice that can hold either an integer or a real value. The ChoiceElement template provides an assignment operator that creates a new object of type T, assigns the value, and sets the discriminent to the integer specified in the template. Therefore, an assignment such as

```
class my_choice : public CHOICE {
public:
        ChoiceElement<INTEGER, 0> i;
        ChoiceElement<REAL, 1>    x;

        my_choice() : i(this), x(this) {}
};
```

Figure 3.14 - Example CHOICE

```
my_choice mc;
mc.i = 10;
```

calls the assignment operator *ChoiceElement<INTEGER, 0>::operator=(INTEGER& v)*, which calls *CHOICE::destroy()* to delete the previous value if it exists, sets the type in mc to 0, creates a new INTEGER, and assigns the value 10 to that integer.

Encoding a CHOICE is simple since the type discriminent is also the subscript of the cons array. The CHOICE encoder simply calls

```
cons[type]->encode(pe);
```

which calls

```
ChoiceElement<INTEGER, 0>::encode(MetaPE* pe)
```

since an integer is currently stored, which in turn invokes the INTEGER encoder to translate the value currently stored in the CHOICE.

Decoding is also straightforward since it is simply a matter of comparing the type information of each ChoiceElement to the type information of the encoded form. When a match is found, the discriminent is known, and the proper decoder can be invoked. The for loop in the decode routine shown in Figure 3.12 performs the type comparison, followed by a call to the decode function. Note that since the ChoiceTag method is static, it is not necessary to maintain an instance of each type in order to perform the type comparisons.

3.2 Transfer Syntax Class Hierarchy

The transfer syntax class hierarchy provides the data translation services required to share data structures. The class hierarchy is structured so that it presents the same external interface, regardless of the transfer syntax specified, so the data representation classes can call the proper encoding routines to apply any transfer syntax. The uniform external interface is a crucial element of the design, since it is the uniform external interface that permits the separation of representation and translation. To present the same external interface, the PE class is defined as a C++ template with a parameter that specifies the encoding rules to be used. The MetaPE class is introduced as a superclass of the PE class to allow the data representation classes to refer to any PE, regardless of its encoding rules. Figure 3.15 shows

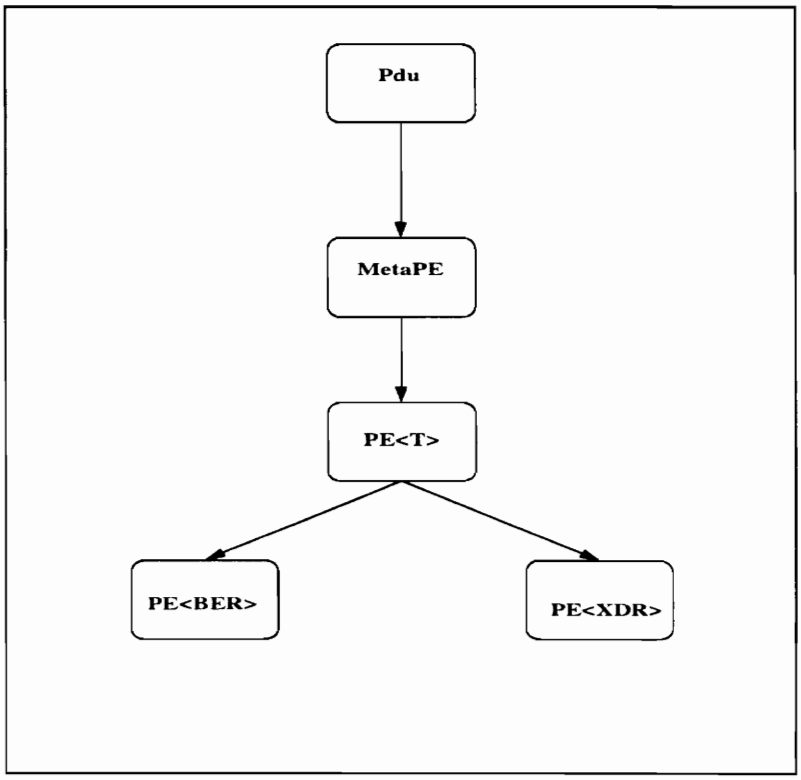


Figure 3.15 - Translation Class Hierarchy

the structure of the transfer syntax class hierarchy. The Pdu class provides the data structures and methods necessary to construct a presentation layer protocol data unit. This class was developed as part of the OOSI project [Laven93] and will not be discussed here.

The link between the representation and translation class hierarchies occurs in the constructors and assignment operators of the representation classes with MetaPE parameters and the constructors and assignment operator of the translation classes with AsnType parameters. Note that a PE class instance is actually passed to the representation class, but the MetaPE abstraction allows the representation class to refer to any PE (PE<XDR> or PE<BER>). The AsnType class performs the same function for the translation classes. The definition of the MetaPE class is shown in Figure 3.16.

A representation class constructor that takes a MetaPE argument, builds a representation from the PE passed by calling the method in the PE class that decodes the data type of the representation class. Similarly, an assignment operator that takes a MetaPE argument assigns a value to the representation based on the PE by having the PE decode the next value. Conversely, the PE constructor and assignment operator with AsnType parameters, construct and assign a value to a PE from an AsnType by calling the AsnType's encode method, which invokes the proper encoding routine in the PE. Since each representation class contains an encode method, the translation class constructor and assignment operator does not need any information about the

representation class. The C++ virtual function handling ensures the encode method of the correct class is called. Therefore, the encoding and decoding of data structures can be viewed as a form of type conversion, since the constructors and assignment operators perform conversions between the internal form of a data structure and its representation according to the specified transfer syntax.

```
class MetaPE : public Pdu {
public:
    MetaPE() : Pdu(PDU_LEN) {}
    MetaPE( octet* data, int len) : Pdu(data, len) {}

    /* Primitive type encoders/decoders */
    virtual int prim2pe(const AsnBool&, const AsnTag&) = 0;;
    virtual int pe2prim(AsnBool&, const AsnTag&) = 0;

    virtual int prim2pe(const AsnInt&, const AsnTag&) = 0;
    virtual int pe2prim(AsnInt&, const AsnTag&) = 0;

    virtual int prim2pe(const AsnReal&, const AsnTag&) = 0;
    virtual int pe2prim(AsnReal&, const AsnTag&) = 0;

    virtual int prim2pe(const String&, const AsnTag&) = 0;
    virtual int pe2prim(String&, const AsnTag&) = 0;

    virtual int prim2pe(const BitString&, const AsnTag&) = 0;
    virtual int pe2prim(BitString&, const AsnTag&) = 0;

    virtual int prim2pe(const AsnNull&, const AsnTag&) = 0;
    virtual int pe2prim(AsnNull&, const AsnTag&) = 0;

    virtual int prim2pe(const AsnObj&, const AsnTag&) = 0;
    virtual int pe2prim(AsnObj&, const AsnTag&) = 0;

    /* Constructed type encoder/decoder */
    virtual int cons2pe(const AsnTag&, int n = -1) = 0;
    virtual int pe2cons(const AsnTag&) = 0;

    /* Type tag manipulation */
    virtual int tag2pe(const AsnTag&) = 0;
    virtual int pe2tag(const AsnTag&) = 0;
    virtual int cmptag(const AsnTag&) = 0;
    virtual void start() = 0;
    virtual void end() = 0;
};
```

Figure 3.16 - MetaPE

One important design decision was to overload the translation methods and name all the functions that perform encoding of primitive data types *prim2pe* and all functions that decode primitive data types *pe2prim.* Defining a prim2pe and pe2prim for AsnBool, AsnInt, and AsnReal allows the template class AsnPrim to call the correct functions based on the type argument supplied to the template rather than containing code specific to each type. The C++ compiler generates code to call the correct function based on the types of the arguments. Two methods, *cons2pe* and *pe2cons* are defined to encode and decode information at the beginning of a constructed type.

In addition to the overloaded functions in the PE class, the virtual functions *encode* and *decode*, that must be implemented in each representation class, serve two important purposes. First, these methods allow the constructor and assignment operator of the PE class to call the correct encode function, without having any detailed type information about the object being encoded. To clarify how this is accomplished, consider the simple example of constructing a PE from an INTEGER using the BER transfer syntax.

```
INTEGER     i = 10;
PE<BER>     pe = i;
```

After the normal sequence of base class constructors for the PE class has completed, the body of PE<BER>::PE(AsnType& asn) is executed. First the start method is called to

perform some initialization of the Pdu. Next, asn.encode(this) is executed. Since asn is a reference to the INTEGER i, INTEGER::encode(MetaPE* pe) will be called.[1] The encode method then invokes pe->prim2pe(int i, AsnTag t), which is a call to the BER integer encode routine since pe is a pointer to a PE<BER> type. The prim2pe method performs the actual encoding of the integer value, along with any type information that needs to be encoded with the value.

The second purpose of these virtual functions is to allow constructed types to encode and decode all elements in the same manner, without any type information for individual elements. For example, to encode a SEQUENCE, first the constructed type information must be encoded, then each element of the SEQUENCE must be encoded. The constructed type information is encoded by calling the cons2pe method at the start of every SEQUENCE encoding. In the case of XDR, where no constructed type information is needed, cons2pe simply returns. The SEQUENCE encode method then executes a for loop to call the encode method for each of its elements. If the element is a primitive type, the steps described in the example above are performed. If the element is a constructed type, the encode method is recursively called.

[1]Actually, AsnPrim<int, UNIVERSAL, 2>::encode(MetaPE* pe) will be invoked since INTEGER is a typedef for AsnPrim<int, UNIVERSAL, 2>.

4.0 Performance Analysis

The previous chapters have shown how simple the ASN class hierarchy is to use for building shared data structures and how simple it is to translate those data structures using either the XDR or BER encoding rules. However, there is usually a tradeoff between flexibility and performance. This chapter presents a performance comparison of the translation facilities provided by the ASN class hierarchy and those provided by the Sun XDR library routines and the ISODE generated BER routines. A variety of data structures are used to exercize different elements of the representation and translation schemes, to be representative of different applications, and to prevent one implementation from having an advantage over the others. Section 4.1 describes the structure of the experiments, followed by the results in Section 4.2. Finally, conclusions are drawn in Section 4.3.

4.1 Structure of the Experiments

To compare the translation times of the same data structure using the BER and XDR encoding rules and the different implementations of those rules, it is necessary to chose data structures that can be translated by both the XDR and BER rules. Specifically, this eliminates sets and bit strings since XDR has no way of representing these types. However, two constructed types (SEQUENCE and SEQUENCE_OF) as well as two primitive types (INTEGER and OCTET_STRING) were tested to determine their relative performance. The structure of these experiments is similar to those of [Bran92] and [Huit92]. Sets were compared separately, using only the C++ and ISODE implementation of the BER encoding rules.

Integers were chosen to test the time required to translate simple primitive data elements. Since the BER encoding rules encode an integer value using the smallest number of bytes possible, it is necessary to vary the value of the integer being encoded so that BER encodings do not enjoy an unfair advantage over XDR encodings. To achieve this, an array of 50,000 integers[1] is constructed an initialized with values from 0 to 5,000,000. To obtain the encoding time, a timer is started, a for is loop is executed to encode each element of the array, then the timer is stopped. Since the encoded forms of

[1] A C array is used, rather than a SEQUENCE_OF.

the 50,000 integers are now in the presentation element, it is quite simple to obtain the decoding time. A pointer in the PE class is reset, the timer started, a for is loop executed 50,000 times to decode each element of the array, then the timer is stopped. This experiment is repeated 50 times, with the average presented in Section 4.2.

To obtain a translation time for the string primitive types, 50,000 octet strings, varying in length from 0 to 30, are translated. The length of 30 was chosen to facilitate comparison with the experiments conducted by Huitema and Chave [Huit92]. Again the experiment is repeated 50 times with the average presented in Section 4.2.

The SEQUENCE constructed type was evaluated using a sequence consisting of one integer and one string which is translated 50,000 times. The integers are initialized with the same values used for the integer test and the strings are initialized with the same values as the string experiment. Each time a sequence is encoded, one integer and one string is encoded. The experiment consists of encoding 50,000 sequences, and is repeated 50 times.

The SEQUENCE_OF constructed type is evaluated by encoding a sequence of 50,000 integers. The array elements are given the same value used for the integer experiment. A single sequence of 50,000 integers is encoded and decoded to facilitate comparison with the integer experiment.

The CHOICE metatype was evaluated using a choice of one integer or one string. As in the other experiments, an array of 50,000 elements is used. Every array element with an even subscript is valued with a integer, while the odd subscripts are given strings. The integer values range from 0 to 5,000,000 and the strings vary in length from 0 to 30.

Finally, the SET type is tested using a set of one integer and one string. Although the XDR rules cannot represent a set, this test is importent since the set type is used frequently in protocols defined with ASN.1 [Huit92]. The integers and strings of the sets are valued as in the sequence test and 50,000 of these sets are translated using the C++ implementation of the BER encoding rules and the ISODE implementation of the BER encoding rules.

4.2 Results

The results of the performance experiments described in Section 4.1 are shown in Figure 4.1. Each experiment was compiled using the GNU g++ compiler, version 2.4.5, with the -O option, and run 50 times on a lightly loaded Sun 4 workstation. Note that the times shown in Figure 4.1 are the average for encoding and decoding 50,000 of the specified data structures and the units are seconds.

Data Structure	Sun XDR		C++ XDR		C++ BER		ISODE BER	
	encode	decode	encode	decode	encode	decode	encode	decode
INTEGER	0.102682	0.091236	0.226094	0.220698	0.343465	0.367502	1.995854	1.983612
STRING	0.667287	0.464178	0.650966	0.699339	0.565286	0.700003	2.658952	2.35231
SEQUENCE	0.928109	0.573974	1.072857	1.27492	1.303438	1.735562	7.347627	6.073607
CHOICE	0.627949	0.456684	0.917373	0.787346	0.604038	0.916149	3.797492	4.003197
Seq_of	0.142731	0.37936	0.250065	0.312292	0.369075	0.510342		
SET					1.309154	4.218824	7.328256	6.422732

Figure 4.1 - Performance Test Results

The results of each individual experiment are shown graphically in Figures 4.2 through 4.7 on the following pages. The SEQUENCE_OF test does not have a value for translations using the ISODE BER encoding rules since the encoding times increased exponentially and it was not feasible to perform this test with 50,000 elements. With only 10,000 elements, the encoding time was over one minute. This indicates that the ISODE SEQUENCE_OF encoding algorithm should probably be rewritten.

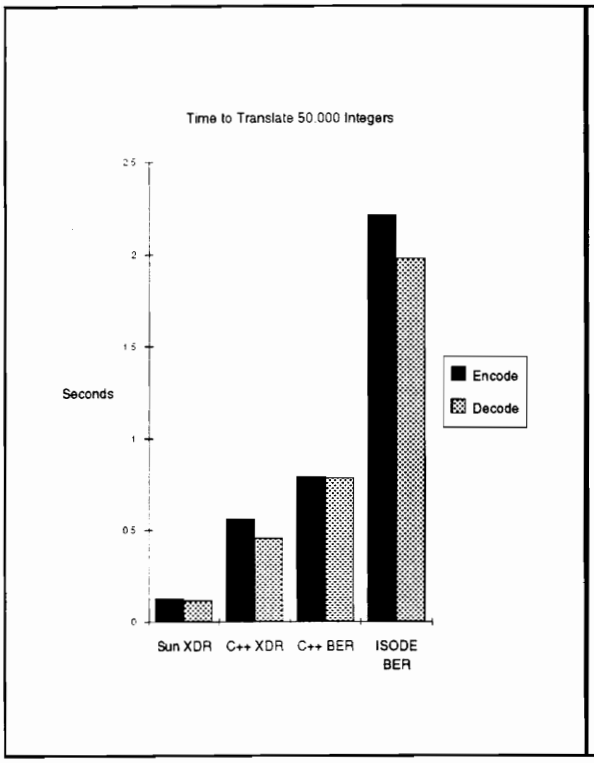


Figure 4.2 - Integer Performance Test

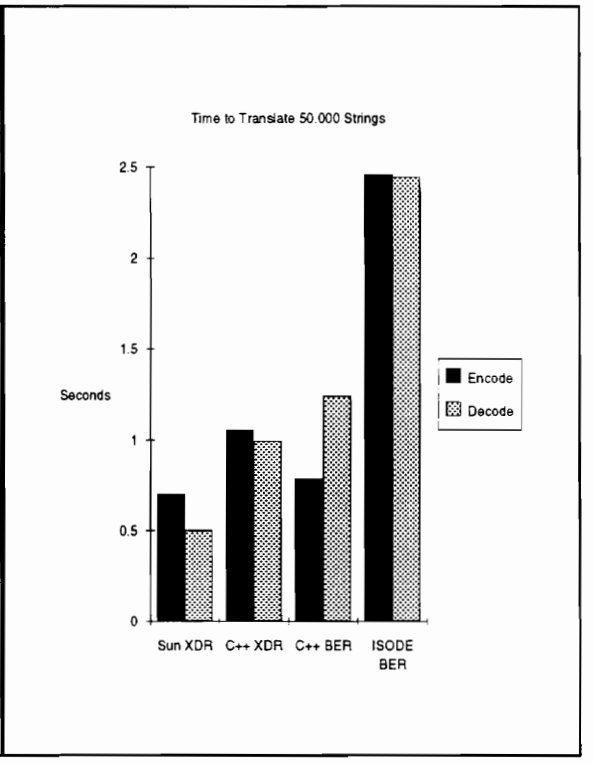


Figure 4.3 - String Performance Test

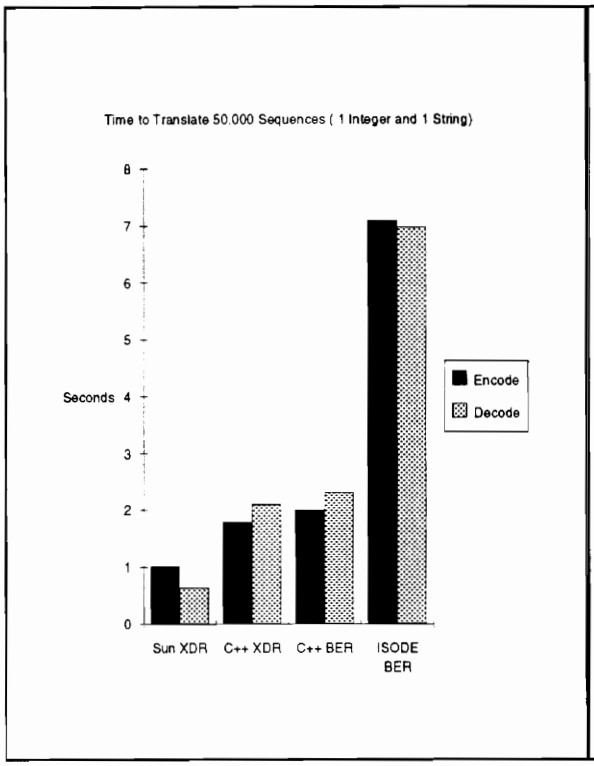


Figure 4.4 - Sequence Performance Test

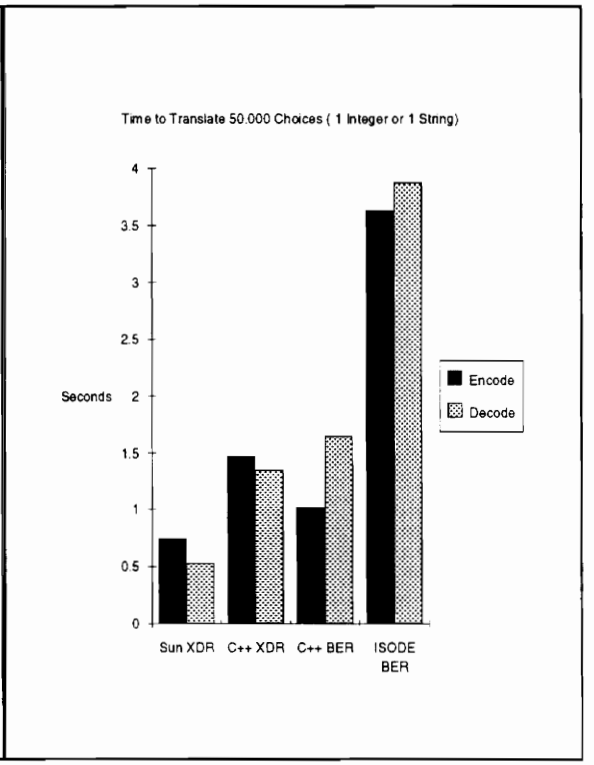
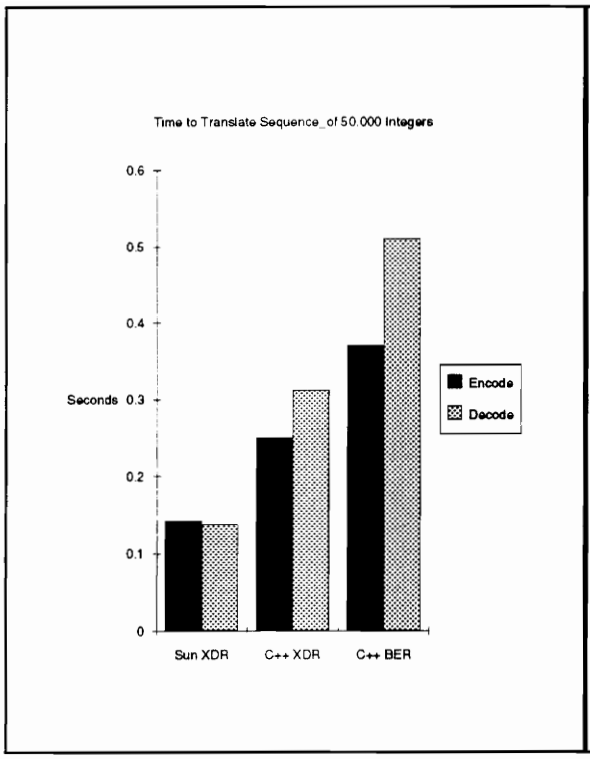


Figure 4.5 - Choice Performance Test



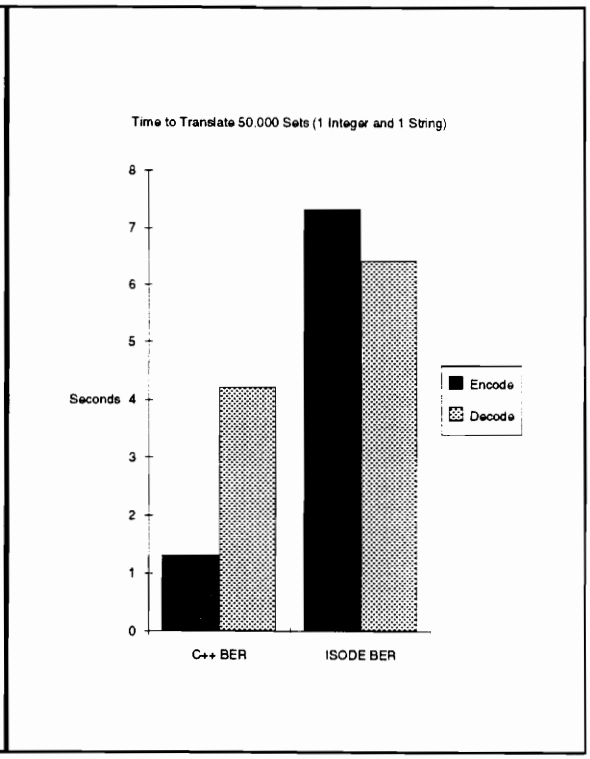


Figure 4.6 - Sequence Of Performance TestFigure 4.7 - Set Performance Test

4.3 Conclusions

As previous comparisons of Sun's implementation of XDR and ISODE's implementation of BER have shown, the XDR encodings are up to 20 times faster than the BER encodings [Huit89b]. The ASN class hierarchy, while more flexible than either

Sun's XDR or ISODE's BER, falls between them when ranked by performance. Since there is almost always a tradeoff between functionality and performance, it is not surprising that the ASN class hierarchy did not perform as well as the Sun's implementation of XDR. However, it may be surprising to some that the ASN class hierarchy BER translations performed better than the ISODE BER translations in every test and the greatest difference between the C++ versions of the XDR and BER encoding rules showed the XDR rules to be less than twice as fast as the BER rules.

Memory allocation for the storage of the encoded data structures presents a difficult problem when comparing the various implementations. With Sun's XDR and the C++ implementations of XDR and BER, it is possible to allocate a large buffer for the encoded data structure before starting the performance tests. In fact, the current implementationof the Pdu class will not dynamically expand the size of the buffer, and therefore requires the allocation of a buffer large enough to contain the entire encoded data structure when the Pdu is initially constructed. However, ISODE creates a complex data structure to store the encoded data structure which must be allocated during the translation process, and the allocation cannot be separated from the translation. Therefore, it is importent to remember that the performance tests are somewhat biased against ISODE, since all of the memory allocation could not be factored out of the ISODE tests. It should be noted that this is one of the disadvantages of using a complex data structure

to represent the encoded data, since the other systems can simply allocate more memory, while ISODE must expand its PE data structure and value all of its members.

Sun's version of XDR performs better than the other implementations since it enjoys several advantages. First, there is no overhead associated with providing a more useful user interface. Second, the code does not have to portable to other hardware platforms, while the C++ class hierarchy and ISODE are both portable. Sun may also perform word-oriented read and write operations, taking advantage of the byte ordering within a word on Sun machines, while the other implementations cannot make such assumptions.

Beside the fact that ISODE is portable to many platforms and cannot take advantage of word-oriented operations, another reason for its slower performance may be the complex data structures used when representing encoded data that must be traversed when encoding and decoding.

Since the C++ class hierarchy imposes the same amount of overhead for both the XDR and BER encoding rules, it is possible to make a fairer comparison between the two. Specifically, when comparing Sun's XDR to ISODE's BER, it is difficult to factor out the memory allocation and deallocation and complex data structure initialization and traversal to simply compare the speed of the encoding rules. With the C++ implementations of the

XDR and BER encoding rules and Sun's XDR rules, it is possible to structure the performance tests so that no memory allocation is necessary, which is impossible to do with the ISODE tests. Therefore, it is important to remember that the ISODE tests are performing more work than just the translation of the data, and the other implementations would also have to perform some of this work to allocate memory to hold the encoding of large data structures.

Using the C++ class hierarchy translation times, several conclusions may be drawn about the encoding rules. First, the integer translation experiment shows that XDR excels at translating fixed length data types. Second, the string translation experiment shows the variable length encoding scheme of the BER is faster when translating variable length data such as strings.

Repeating the same experiment with a sequence consisting of an integer and a string with the same values determines the amount of overhead associated with translating a constructed type. If the same number of sequences are translated, the best time this experiment could possibly obtain is the sum of the integer translation time and the string translation time. Any time longer than this must be attributed to the overhead of translating a constructed type. The amount of translation time considered overhead is calculated by dividing the difference between the actual sequence translation time and the ideal sequence translation time by the actual sequence encoding time. The amount of

overhead associated with a sequence_of can be determined by encoding a sequence_of 50,000 integers and comparing the result with encoding 50,000 individual integers. These results are shown in Figure 4.8.

	Sun XDR		C++ XDR		C++ BER		ISODE BER	
	encode	decode	encode	decode	encode	decode	encode	decode
Seq Time	0.769969	0.555414	0.87706	0.920037	0.908751	1.067505	4.654806	4.335922
% Overhead	17.03895	3.233596	18.25006	27.83571	30.28046	38.49226	36.64885	28.61043
Seq_of Time	0.142731	0.37936	0.250065	0.312292	0.369075	0.510342		
% Overhead	28.05908	75.95002	9.585908	29.3296	6.938969	27.98907		

Figure 4.8 - Constructed Type Overhead

The choice performance test shows the ASN class hierarchy XDR rules taking 45% longer to encode a choice than the ASN class hierarchy BER rules. This can be attributed to two factors. First, the XDR rules were slower when encoding strings, which constituted half of the data in the choice experiment. Second, XDR rules for encoding a choice require an integer discriminant be encoded before the value of the choice. With the XDR rules, this is handled by separately encoding the discriminant, followed by encoding the choice. The BER rules have a discriminent included with every encoding, and, therefore, do not require a separate encoding.

5.0 Conclusion

This project demonstrated that it is possible to separate the representation of a shared data structure from its translation. The goals of allowing flexible encodings and allowing the user to control the names of user defined types and identifiers have been achieved. Flexible encodings are realized by defining a general translation class that maintains a uniform external interface, regardless of the transfer syntax specified. User control of names is accomplished by using the classes defined for data representation in the ASN class hierarchy. The validity of the encodings generated by the translation classes has been demonstrated by interoperating with servers written using Sun Microsystems rpcgen compiler and ISODE's ASN.1 compiler and runtime environment. The simple performance analysis showed the speed of encodings performed by the class library to lie between Sun's XDR encodings and ISODE's BER encodings.

This project leads the way for a great deal of future work. First, a large number of protocol specifications are written using ASN.1 notation. While it is quite simple to manually translate these specifications to ASN class specifications, it is clear that a translator could be written to perform this task. This project is currently in progress. A

related project may be to develop a translator for specifications written for rpcgen compiler.

Another area that needs additional research is that of performance. Certainly, a more detailed performance analysis could be conducted, perhaps with comparisons to work done by Huitema, Doghri, and Chave [Huit92] to develop a high performance light weight transfer syntax. The results of this analysis could be used to identify areas needing performance optimizations.

While the class hierarchy provides representation and translation services, it is still necessary to transport the encoded data from one system to another. Services to perform this function are provided at layers below the presentation layer. A project is under way to integrate the ASN class hierarchy with the OOSI project developed by Kafura and Lavender [Laven93]. OOSI currently provides session and transport layer services, but needs the presentation layer services of the ASN class hierarchy to provide a useful service to users. After the ASN class hierarchy is integrated with OOSI, it will be simple to define OSI protocols and to provide an OSI environment that is easier to use than ISODE.

The error processing of the ASN class hierarchy is currently somewhat limited. It would be a useful extension to this project to add C++ exception handling. When an

error condition is encountered, the function *throws* an exception which may be caught and handled by the user, or if not caught by the user, will cause the program to terminate. This task becomes more complicated when the OOSI project is included, since exceptions will have to be caught as well as thrown. For example, the quality of service associated with a transport connection may deteriorate. At this point, the transport entity would throw an exception, indicating an error condition. The session entity should catch the exception and take appropriate action based on the service request of the user. The connection may continue to be used in its degraded state, the session entity may opt to terminate the current transport connection and establish a new transport connection or the session connection may be aborted, and an exception thrown for the session service user.

Hopefully this project and the related projects discussed in this section will help to ease the daunting task of using OSI protocols and help to make their use more wide spread.

References

[Abbot92] Mark B. Abbot, Larry L. Peterson. A Language-Based Approach to Protocol Implementation. *Computer Communication Review*, Volume 22, Number 4, October, 1992.

[Boch90] Gregor v. Bochman, Michel Deslauriers. Combining ASN.1 Support with the Lotos Language, Protocol Specification. *Testing, and Verification*, IX, 1990.

[Bran92] M. L. Branstetter, J. A. Guse, D. M. Nessett. ELROS - An Embedded Language for Remote Operations. *Upper Layer Protocols, Architectures and Applications*, 1992.

[Carr89] Nicholas Carriero, David Gelernter. Linda in Context. *Communications of the ACM*, Volume 32, Number 4, April, 1989.

[Huit89a] Christian Huitema, Assem Doghri. Defining Faster Transfer Syntaxes for the OSI Presentation Protocol, *ACM Communication Review*, Volume 19, Number 5, October 1989.

[Huit89b] Christian Huitema, Assem Doghri. A High Speed Approach for the OSI Presentation Protocol. *IFIP*, 1989.

[Huit92] Christian Huitema, Ghislain Chave. Measuring the Performance of an ASN.1 Compiler. *Upper Layer Protocols, Architectures and Applications*, 1992.

[Jain93] Bijendra N. Jain, Ashok K. Agrawala. *Open Systems Interconnection*, McGraw-Hill, New York, NY, 1993.

[Kaf92] Dennis Kafura, Greg Lavender, Rajesh Khera. A Class Hierarchy for Data Translation in a Heterogeneous Computing Environment. Dept. of Computer Science, VA Tech, 1992.

[Koiv91] Juha Koivisto, J. Malka, J. Reilly. An Object-Oriented Approach to Distributed Computation, *UNENIX C++ Conference Proceedings*, Washington D.C., April 1991.

[Laven93] R. G. Lavender. *Polymorphic Types for Constructing Concurrent Object and Layered Communication Protocols*, PhD Dissertation, Dept. of Computer Science, VA Tech, May 1993.

[Lea92] Doug Lea. *User's Guide to the GNU C++ Library*, Free Software Foundation, Inc., Cambridge, MA, April 29, 1992.

[Li89] Kai Li, Paul Hudak. Memory Coherence in Shared Vitrual Memory Systems. *ACM Transactions on Computer Systems*, Volume 7, Number 4, November 1989.

[Maek87] Mamoru Maekawa, Arthur E. Oldehoeft, Rodney R. Oldehoeft. *Operating Systems Advanced Concepts*, The Benjamin/Cummings Publishing Company, Meno Park, CA, 1987.

[Neu90] Gerald W. Neufeld, Yueli Yang. The Design and Implementation of an ASN.1-C Compiler. *IEEE Transactions on Software Enginerring*, Volume 16, Number 10, October 1990.

[Nicol93] John R. Nicol, C. Thomas Wilkes, Frank A. Manola. *Object Orientation in Heterogeneous Distributed Computing Systems,*

[Prasad93] Raja Prasad, Ulloa Gonzalo. A simple Encoder for Fieldbus Applications. *Computer Communication Review*, Volume 23, Number 1, January, 1993.

[Rose90] Marshall T. Rose. *The Open Book: A Practical Perspective on OSI.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[Rose91a] Marshall T. Rose. *The ISO Development Environment Users Manual*, Volumes 1 and 4, Performance Systems International, July 1991.

[Rose91b] Marshall T. Rose. *The Simple Book*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[Stall87] William Stallings. *Handbook of Computer Communications Standards*, Volume 1, Macmillan Publishing Company, New York, NY, 1987.

[Steed90] Douglas Steedman. *Abstract Syntax Notation one (ASN.1) The Tutorial and Reference*, Technology Appraisals, 1990.

[Stev90] W. Richard Stevens. *UNIX Network Programming*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[Sun88] Sun Microsystems. *Network Programming*, May 1988.

Appendix A

ASN Class Library User's Manual

This chapter introduces the details of a C++ class library that contains classes used for building data structures that can be shared in a distributed, heterogeneous environment. The classes can be grouped into two categories, data representation, also called the Abstract Syntax Notation (ASN) category, and data translation. The data representation category provides classes for representing primitive data types such as integer and real, and mechanisms to combine the primitive types to form arbitrarily complex data structures. These data structures are then translated to a standard network format using translation classes that implement the encoding rules of the Basic Encoding Rules (BER) defined by the ISO, or the External Data Representation (XDR) rules defined by Sun Microsystems. A Presentation Element (PE) is a class that encapsulates the encoding rules and translation specific elements.

The following data representation topics will be discussed:

- Primitive data types
  - Simple types - INTEGER, REAL, BOOLEAN
  - String types - Character strings, OCTET_STRING, BIT_STRING
  - OSI types - OBJECT_IDENTIFIER
- Constructed types
  - SEQUENCE
  - SEQUENCE_OF
  - SET
  - SET_OF
- CHOICE
- ANY

The final sections of the chapter will discuss data translation topics, and provide some information specific to the BER encodings. The following topics will be presented:

- Changing and adding type information
- Presentation Element

Primitive Data Types

The ASN class library contains three categories of primitive data types. The simple primitive types are INTEGER, REAL, and BOOLEAN which intuitively correspond to int, double, and enum {FALSE, TRUE} in the C programming language. The string category of primitive types corresponds to the char* representation in C with the appropriate functions such as copy and concatenation implemented in library routines. The third primitive classification contains only the object identifier type which has no analogous C type. All of the data types discussed in this section are defined in the ASN.h header file which must be included in any programs using the ASN class library to build shared data structures.

Simple Primitive Types

The simple primitive data type definitions can be used interchangeably with the built in data types int, double, and enum {FALSE, TRUE}. The declarations of variables of these types are shown in the following code fragment:

```
INTEGER   i;
REAL      x = 1.5;
BOOLEAN   done(TRUE);
```

which declares a variable *i* of type INTEGER, a variable *x* of type REAL with the value 1.5, and a variable *done* of type BOOLEAN with the value TRUE. These variables may be used in expressions wherever it is syntactically correct to use variables of their corresponding built in type. However, it is recommended that programmers only use these classes for data that is to be shared, due to the overhead of accessing the values through the C++ class mechanisms. The built in data types should be used for loop counters and other heavily accessed data items internal to a program.

Each of these classes is a specific type of the AsnPrim template class, each with the same external interface shown in Figure A.1.

```
template <class T> class AsnPrim {
protected:

    T   prim;   /* primitive value */

public:

    int primitive() const { return 1; }

    const AsnTag& tag()             { static AsnTag t(C, PRIM_FORM, I); return
                                    t; }
    const AsnTag& ChoiceTag()       { static AsnTag t(C, PRIM_FORM, I); return
                                    t; }

    inline AsnPrim()                {}
    inline AsnPrim(T& v)            { prim = v; }
    inline AsnPrim(AsnPrim& p)      { prim = p.prim; }
    inline AsnPrim(MetaPE& pe)      { pe.start(); pe.pe2prim(prim, tag()); }

    inline void operator=(T& v)     { prim = v; }
    inline void operator=(AsnPrim& p) { prim = p.prim; }
    inline void operator=(MetaPE& pe){ pe.start();
                                       pe.pe2prim(prim, tag());}

    inline operator T() const       { return prim; }

    int encode(MetaPE* pe)          { return pe->prim2pe(prim, tag()); }
    int decode(MetaPE* pe)          { return pe->pe2prim(prim, tag()); }
    void print (int tab = 0);
};
```

Figure A.1 - Primitive Type Interface

The simple primitive types are defined with the typedef statements shown below:

```
typedef  AsnPrim<int>       INTEGER;

typedef  AsnPrim<double>    REAL;
```

```
typedef  AsnPrim<bool_t>   BOOLEAN;
```

As instantiations of the AsnPrim class, the INTEGER, REAL, and BOOLEAN classes each contains a set of constructors and assignment operators that take the same AsnPrim type or the associated built in type, and a type cast operator to convert the AsnPrim type to it's built in type. These methods allow simple primitive type variables, such as INTEGER, to be used interchangeably with variables of the built in types, such as int. In addition, the ASN-related type information may be obtained using the *tag* and *primitive* methods. The *encode* and *decode* methods take a pointer to a *MetaPE* object, which is a superclass of all *Presentation Element* (PE) classes, and call the appropriate method of the PE to convert from the internal representation of the primitive value to the network representation, or vice versa. The use of these PE classes is explained later in this chapter. Since the argument to the encode and decode methods is a MetaPE, any type of PE, whether it employs XDR or BER encoding rules, may be passed, thereby allowing a data structure to be translated using any of the defined set of encoding rules.

A *print* method is provided for each class which displays the type information and the value of the object in a user friendly format. This method, available in every data representation class, is useful for debugging purposes. The following example

```
INTEGER i(10);
i.print()
```

produces output

```
( UNIV INT
        10
)
```

Information such as the class of the object, in this case UNIV (for universal, a term defined in the ASN language), and the type, in this case INT, is only used by the BER encoding rules, but is available for all data types independent of the rules used to encode the data.

String primitive types

The ASN string types provide implementations of character strings. These classes provide useful operations for manipulating the strings as well as methods that allow the encoding and decoding of the data. The character string types and their associated character sets are listed in Figure A.2.

NumericString	0 - 9, space
PrintableString	a - z, A - Z, 0 - 9, ' " ( ) + , - . / : = ? and space
TeletexString	CCITT T.61 character set
VideotexString	CCITT T.100 and T.101 character sets
IA5String	all characters of IA5
VisibleString	visible characters of IA5 and space
GraphicString	all ISO registered graphic characters and space
GeneralString	all ISO registered graphic and control characters, space, and delete

Figure A.2 - String Types [Steed91]

The character string types of the ASN class library match the string types defined in the ASN.1 standard. The general string type, OCTET_STRING, is composed of arbitrary eight bit character values. Since there are no restrictions on the values that can be placed in each character, each character or octet can take any value from 0 to 255. The other strings are subtypes of OCTET_STRING that only allow individual characters of the string to take values from a specified character set. For example, a valid NumericString type allows it's character to take only the values 0 through 9, and space. The ASN class library string types do not enforce these restrictions due to the processing overhead that would be incurred, however, such enforcement could easily be added.

The string classes are implemented with the GNU libg++ string class which provides a set of useful operators. While it is possible to use the primitive string types as if they were of type char*, the operators allow easier programming and provide a greater degree of type checking. Some of the more common methods available for strings are listed in Figure A.3. For a more detailed discussion, see [Lee92].

lexicographic relational operators	==, !=, <, >, <=, >=
compare(String, String)	compares the two strings
fcompare(String, String)	compares the two strings with case insensitivity
length()	returns the length of the string
at (int pos, int len)	returns the substring of length len starting at position pos
+, +=	string concatenation
cat (x, y, z)	string concatenation, z = x + y
reverse (String)	returns the reverse of String
upcase (String)	returns String with all letters converted to upper case
downcase (String)	returns String with all letters converted to lower case
capitalize (String)	returns String with first letter of each word in String capitalized

Figure A.3 - String Operators [Lee92]

It should be noted that since this version of the ASN class library does not enforce any character value restrictions, all the string types, including OCTET_STRING, are effectively equivalent.

An example using the string classes is shown below.

```
OCTET_STRING     s1, s2 = "Virginia Tech ";
s1 = "Computer Science";
s1 = s2 + s1;                    // concatenation
PE<XDR> pe = s1;                 // encode using XDR rules
OCTET_STRING new_str = pe;       // decode using XDR rules
cout << (char*) new_str;         // type cast required
```

This example shows the declaration of two OCTET_STRING variables, s1 which has no value, and s2 which is assigned the string "Virginia Tech." The variable s1 is then assigned "Computer Science." These assignments are possible since the string classes are subclasses of the GNU libg++ String class which provides constructors and assignment operators and dynamically allocates space when a value is assigned to a string variable. The next statement assigns the concatenation of s2 and s1 to s1, producing the string "Virginia Tech Computer Science." As with all other ASN types, the string types can be encoded and decoded into a presentation element as shown in the next two

statements, which translate the string using the XDR encoding rules. Finally, the string can be printed as shown by the last statement.

The BIT_STRING class implements a type that allows the manipulation of an arbitrarily long string of bits (zeros and ones). Using the BIT_STRING class is similar to the character string classes except the values assigned must be zero or one. For example,

```
BIT_STRING      b = "1001000111";
```

Set operations	~, &, &=, |, |=, -, ^, ^=
Relational operators	==, !=, <=, <, >=, >
set()	set all bits
set (n)	set bit n
set (m, n)	set bits m through n
clear()	clear all bits
clear (n)	clear bit n
clear (m, n)	clear bits m through n
invert()	invert all bits
invert (n)	invert bit n
invert (m, n)	invert bits m through n
test(n)	returns true if bit n is set
test (m, n)	returns true if any of bits m through n are set
first() or first(1)	returns index of first set bit
first(0)	returns index of first clear bit
next(n, 1) or next(n)	returns index of next set bit after position n
last (1)	returns index of rightmost set bit
previous (n, 0)	returns index of previous clear bit before the nth position

Figure A.4 - BitString Operators [Lee92]

creates a BIT_STRING variable *b* and assigns a sequence of 10 bits to it. The BIT_STRING class is implemented using the GNU libg++ BitString class which contains a set of operators useful for manipulating strings of bits. Figure A.4 lists a number of the more useful operations available for using bit strings.

Object Identifier

The object identifier type, which is used in ISO and CCITT standards to represent nodes in the object identifier tree (OIT), has no analogous type in the C language. While an object identifier is denoted by a list of dot separated non-negative integers, the semantics of the object identifier are quite different from an array of integers. Each node of the OIT represents a registered information object such as a standards document, a module specification, or a remote operation. An internal node can be used to represent a standard with it's children representing specific parts of the standard, and administrative responsibility can be delegated for the various subtrees of an internal node [Rose90]. The integers used to represent object identifiers indicate which branch should be taken during tree traversal, starting from the root. The root node has three children, each of which is administered separately. Child zero of the root is administered by the CCITT, child one is administered by the ISO, and child two is administered jointly by the CCITT and ISO.

Each child may in turn have children of it's own, with administrative responsibility delegated to a specific subgroup of the group responsible for the parent node. For example, the ISO/CCITT directory telephone number attribute is defined as {2, 5, 4, 20} [Steed91].

An object identifier is declared using the predefined class OBJECT_IDENTIFIER, and the values representing the branches of the tree are assigned using a string notation, with each integer separated by a dot. Therefore,

```
OBJECT_IDENTIFIER       oid = "2.5.4.20";
```

declares a variable *oid* of type OBJECT_IDENTIFIER, and assigns to it the value for the ISO/CCITT directory telephone number attribute.

Constructed Data Types

Four classes are provided for combining primitive data types to form more complex data structures: SEQUENCE, similar to a struct in C; SEQUENCE_OF, similar to an array in C; SET, not part of the C language but defined by the ASN.1 standard, is

used to define an unordered collection of elements, each of a different type; and SET_OF, which defines an unordered collection of elements of one type. Since any member of a constructed type may itself be a constructed type, arbitrarily complex data structures can be built. These types are also defined in the ASN.h header file.

Sequence

The SEQUENCE type is an ordered collection of elements, similar to a struct in C or a record in Pascal. The elements may be any combination of the same or different primitive or constructed types. When defining a new sequence type, the new type must inherit publicly from the predefined SEQUENCE class in order to have the necessary data translation methods for sharing the data. The following example shows the almost complete definition for a class named my_sequence containing two integers and one real.

```
class my_sequence : public SEQUENCE {
public:
        INTEGER     i;
        INTEGER     j;
        REAL        x;
};
```

Three details have been omitted from this example. First, the ASN.1 language allows members to be declared optional, meaning the member may or may not have been assigned a value prior to the encoding (or after the decoding) of this data structure. An element with no assigned value is termed an undefined element. The implications for the encoding rules are that undefined optional members do not have to be encoded, and type information will be required to ensure proper decoding of the data structure when one or more elements is not present in an encoding. Second, ASN.1 allows the specification of a default value if a member is not present when decoding. It is important to note that the ASN.1 syntax does not permit both the optional and default modifiers to be specified for a single member. If a member is defined as having a default value, it is implicitly optional. Third, an assignment operator with a MetaPE& argument should be defined that calls SEQUENCE::operator= (MetaPE&) to allow proper decoding when the class is used as a member of a constructed type.

To use the optional and default capabilities, it is necessary to use three new template classes in the definition of a SEQUENCE or SET constructed type, REQUIRED, OPTIONAL, and DEFAULT. The example shown above is modified to show how these templates are used and the assignment operator is included.

```
class my_sequence : public SEQUENCE {
public:
        REQUIRED<INTEGER>    i;
        OPTIONAL<INTEGER>    j;
        DEFAULT  <REAL>      x;

        void operator= (MetaPE& pe) { SEQUENCE::operator=(pe); }
        my_sequence() { x.default_val = 0.0; }
};
```

The example shows a sequence with three elements. The first element is an integer and must always be present. The second element is an integer, but is not required to be defined or present in an encoding of this data structure. The third element is a real that is not required to be present in an encoding of this data type, and if not present, is given the value 0.0 in the data structure that is assigned the decoded value. Those familiar with the XDR rules for encoding data will notice that the XDR rules will not be able to decode this data structure properly if an element is missing from the encoding since the XDR rules do not include type information in the transfer syntax. Therefore, when the user knows that the XDR rules will be used as the transfer syntax, the OPTIONAL and DEFAULT templates cannot be used.

The other detail omitted from the previous sequence examples is how the constructor for the new type must be defined. The details of the implementation of the REQUIRED, OPTIONAL, and DEFAULT templates require their constructors be called with a pointer to the class being defined. Since C++ provides such a pointer, called the

this pointer, defining the constructor is trivial. The complete definition for the class described above is shown below.

```
class my_sequence : public SEQUENCE {
public:
        REQUIRED<INTEGER>    i;
        OPTIONAL<INTEGER>    j;
        DEFAULT  <REAL>      x;

        void operator= (MetaPE& pe) { SEQUENCE::operator=(pe); }
        my_sequence() : i (this), j (this), x (this) {
                x.default_val = 0.0;
        }
};
```

Access to the individual members of the sequence is via the member access operators (. and ->) just as if the data structure were declared a C structure or public members of a regular C++ class. Thus, the code fragment

```
my_sequence  seq;

seq.i = 0;

seq.j = 1;
```

defines a my_sequence variable and assigns the value zero to the member i, and one to the member j. The member x has not been assigned a value, and therefore undefined.

If there is no value for x when decoding, x will be given the default value specified in the constructor.

The print method is also included in constructed types and may be invoked as shown below:

```
seq.print()
```

produces output

```
(UNIV SEQ
        (UNIV INT
                0
        )
        (UNIV INT
                1
        )
        (UNIV REAL
                0.0
        )
)
```

Set

The set type allows the construction of an unordered collection of elements, similar to the set concept in mathematics. A set is constructed in the same manner as a sequence, but inherits from the predefined SET class. In order to allow proper translation of a set data structure, all elements within the set must have a unique type, and the template classes REQUIRED, OPTIONAL, and DEFAULT must be used for each member of the set. For example, the following is not a correct set definition

```
class my_set : public SET {
public:
        REQUIRED<INTEGER>    i;
        REQUIRED<INTEGER>    j;
        REQUIRED<REAL>       x;

        void operator=(MetaPE& pe) { SET::operator=(pe); }
        my_set () : i(this), j(this), x(this) {}
};
```

since the encoder may encode the values in any order, making it impossible to determine which integer value should be assigned to i, and which should be assigned to j.[1] A correct set definition with an integer, real , and a sequence is show below.

---

[1] The ability to change and add type information through implicit and explicit tagging will relax this restriction. These concepts are discussed later in this chapter.

```
class my_set : public SET {
public:
        REQUIRED<INTEGER>           i;
        REQUIRED<REAL>              x;
        REQUIRED<my_sequence>       seq;

        void operator=(MetaPE& pe) { SET::operator=(pe); }
        my_set() : i (this), x (this), seq (this) {}
};
```

Notice that the same constructor convention used for sequence must be followed for set.

A proper encoding of this set can consist of an INTEGER, a REAL, and a my_sequence, in any order. The decoder is responsible for making assignments to the proper members of the set.

Access to members of the set is also through the member access operators (. and ->).

Sequence of

The SEQUENCE_OF type allows construction of an ordered collection of elements of one type, similar to a one dimensional array in many programming

languages. The SEQUENCE_OF may be either fixed or variable length, depending on the constructor that is called. A call to the constructor with an integer size defines a fixed length SEQUENCE_OF, while a call to the constructor with no arguments defines a variable length array whose length can vary dynamically without limit. For example,

```
SEQUENCE_OF<INTEGER>    int_array(10);
SEQUENCE_OF<REAL>       real_array;
```

declares a SEQUENCE_OF ten integers and a SEQUENCE_OF reals that can be any length.

Access to the members of a fixed length or a variable length sequence_of is either with the subscript operator ([ ]), with subscripts starting from zero, or with the member functions listed below.

void start()	-	sets an internal pointer to the beginning of the array
T next()	-	returns the next element in the array and increments the internal pointer
int done()	-	returns non zero if the end of the array has been reached, zero otherwise

The following code fragment defines a SEQUENCE_OF ten INTEGERs, initializes the array, and prints each element.

```
SEQUENCE_OF<INTEGER>        a(10);

for (int i = 0; i < 10; i++)
        a[i] = i;

for (i = 0; i < 10; i++)
        cout << "element " << i << " = " << (int) a[i] << "\n";
```

Set of

The SET_OF type allows construction of an unordered collection of elements of the same type, where the same element may appear more than once and the number of occurences is significant [Steed91]. This is different than a SEQUENCE_OF, where the elements are ordered. This is often referred to as a bag.

Access to members is via the subscript operator ([ ]) or the member functions listed below.

void start() - sets an internal pointer to the beginning of the set of

T next()	-	returns the next element in the set of and increments the internal pointer
int done()	-	returns non zero if there are no more elements, zero otherwise

Choice

The CHOICE type allows construction of a data structure that takes exactly one value from among one or more distinct types, similar to a union in many programming languages. A new CHOICE type must inherit publicly from the predefined CHOICE class, and each member of the CHOICE must have a unique type. The ChoiceElement template is used for each member, providing functionality similar to that provided by the OPTIONAL template. For example,

```
class my_choice : public CHOICE {
public:
        ChoiceElement<INTEGER, 0>       i;
        ChoiceElement<REAL, 1>          x;

        void operator=(MetaPE& pe) { CHOICE::operator=(pe); }
        my_choice() : i(this), x(this) {}
};
```

defines a new type that holds either an integer or a real value. The CHOICE class contains a public integer member named *type* which is also part of the public interface presented by subclasses of CHOICE such as my_choice. The second parameter to the ChoiceElement template is assigned to the *type* member when the CHOICE is assigned a value of the specified type. The *type* member can be tested by the user to determine the type currently stored in the CHOICE. The tag method, which returns an AsnTag, may also be called to compare the type information to the type information of the predefined classes. Members are accessed using the member operators (. and ->) as shown below.

```
my_choice     mc;

mc.i = 1;

if (mc.tag() == INTEGER::tag())
        // integer processing
else
        // real processing
```

Any

The ANY type is a union of all possible types representable in ASN.1, and therefore is similar to a CHOICE. ANY is used when the type is not specified by the

ASN.1 author, such as during the development of a new protocol specification. The value could be further defined later, replacing the ANY specification with a more specific type [Rose90]. Another use for the ANY type is in a protocol specification that contains user defined data, such as an RPC message. For example, a call body of an RPC message could be defined as

```
class call_body : public SEQUENCE {
public:
        REQUIRED<INTEGER>    rpcvers;
        REQUIRED<INTEGER>    prog;
        REQUIRED<INTEGER>    vers;
        REQUIRED<INTEGER>    proc;
        REQUIRED<opaque_auth> cred;   // opaque_auth defined elsewhere
        REQUIRED<opaque_auth> verf;
        REQUIRED<ANY>         user_data;

        void operator=(MetaPE& pe) { SEQUENCE::operator=(pe); }
        call_body() :  rpcvers(this), prog(this), vers(this), proc(this)
                       cred(this), verf(this), user_data(this) {}
}
```

Changing and adding type information

Some transfer syntaxes, such as the BER, include type information as part of the encoded data that is transmitted on the network. In order to avoid ambiguity when decoding a data structure that has optional elements, at times it is necessary to change

or add additional type information to be included in the encoding. The ASN.1 specification refers to this concept as *tagging*. Consider the sequence show below which has two optional integers.

```
class my_seq : public SEQUENCE {
public:
        OPTIONAL<INTEGER>    i;
        OPTIONAL<INTEGER>    j;

        my_seq() : i (this), j (this) {}
};
```

When an encoding for this sequence is received with only one integer, it is impossible to determine if that value should be placed into i or j. ASN.1 provides two possible solutions for this problem. The first solution is to declare the variables i and j to be *implicit* integers, and to specify discriminating type information to be included in the encoding, replacing the integer type information normally encoded. Thus, i and j can be given unique type information, allowing proper decoding when an element is absent from the encoding. The second solution is to add an additional type specification that will be encoded before the normal integer encoding is performed. This *explicit* tagging places the specified tag before the normal integer tag in the encoding. Again, i and j can be

given type information that makes them distinguishable. Our class library includes two templates for changing type information or adding type information to an encoding[2].

```
template class Implicit < type, int class, int number>
template class Explicit < type, int class, int number>
```

The values specified for class are specified in the ASN.1 standard and are listed in Figure A.5. The values specified for number are relatively unimportant as long as the sender and receiver of the information agree on the values.

UNIVERSAL_CLASS	0	Globally unique. This is used for the predefined type information and should not be used by the user.
APPLICATION_CLASS	1	Unique within an application.
CONTEXT_CLASS	2	Unique within the type being defined.
PRIVATE_CLASS	3	Unique within a given enterprise.

Figure A.5 - Tag Classes [Rose90]

[2]The Implicit and Explicit templates will not affect an XDR encoding since XDR does not include type information in it's encoded forms.

The Implicit template directs the encoding routine to use the type information specified for class and number in the encoding rather than the normal type information. For example, a BER encoding of an implicit integer

```
Implicit < INTEGER, CONTEXT, 0>        i;
```

causes the tag field in the encoded form for i to contain the value for CONTEXT and the number 0 rather than the normal integer tag information, UNIVERSAL and 0. The receiver must know that the incoming value is an implicit integer since the normal integer tag information is not present. The print method indicates the type information available during the encoding process as shown in the code fragment and the output produced below.

```
Implicit<INTEGER, CONTEXT, 0>     i = 0;
i.print();
```

```
(CNTX 0
        0
)
```

The Explicit template directs the encoder to encode the type information specified in the declaration before encoding the integer type information as in a normal encoding. For example, an encoding of an explicit integer

```
Explicit < INTEGER, APPLICATION, 1> i;
```

would contain a tag field with APPLICATION and 1 as the type information followed by the normal integer tag field and integer encoding as demonstrated by the example below.[3]

```
EXPLICIT<INTEGER, APPLICATION, 1>      i = 10;
i.print();

(APPL 1
        (UNIV INT
                10
        )
)
```

A solution can now be given for the ambiguous decoding problem discussed at the beginning of this section. For our solution, we choose to make the integers implicit and provide unique tag information for each.

---

[3]Actually, every tag field is followed by a length field specifying the length of the value field in the T-L-V scheme discussed earlier.

```
class my_seq : public SEQUENCE {
public:
        OPTIONAL<Implicit < INTEGER, CONTEXT_CLASS, 0> >   i;
        OPTIONAL<Implicit < INTEGER, CONTEXT_CLASS, 1> >   j;

        my_seq() : i (this), j (this) {}
};
```

If an encoded form of a my_seq is received that contains only one integer, the decoder can determine if the value should be assigned to i or j based on the tag information defined using the Implicit template.

Presentation Element

A presentation element (PE) is an abstraction used to encapsulate the details involved with the transfer syntax and to provide an appropriate data structure that can be passed to the session layer for transmission on the network. The PE class is a template class that is instantiated with a parameter specifying the transfer syntax to be used. Since only the BER and XDR transfer syntaxes have been implemented, there are only two acceptable declarations using the PE class:

```
PE<BER>     ber_pe;

PE<XDR>     xdr_pe;
```

Two header files, XDR.h and BER.h, are provided, one of which must be included in order to define a PE data structure. In addition, the code to perform the translation is in XDR.cc and BER.cc, which must be compiled and linked with the application being developed.

When a data structure defined with the ASN class hierarchy is assigned to a PE object, either in the declaration of the PE or with the assignment operator (=), the data structure is type cast to it's encoded form using the specified encoding rules. For example, both

```
INTEGER     my_integer = 1;
PE<BER>     pe (my_integer);
```

and

```
INTEGER     my_integer = 1;
PE<BER>     pe;
pe = my_integer;
```

cause the encoded form of the INTEGER my_integer to be placed into the Presentation Element pe.

When a PE object is assigned to a data structure, the PE is type cast to the type of the data structure which calls the proper decoding routines, and the values are placed into the appropriate members of the structure. For example,

```
INTEGER     my_integer;
PE<XDR>     pe;
// get data from the network, place it in pe
my_integer = pe;
```

causes the value received from the network to be decoded using the XDR rules and stored into my_integer. Any data structure built using the types of the ASN class library, no matter how complex, can be encoded and decoded using these same simple techniques. For example, the following code fragment defines a new type called my_sequence, assigns values to the members, and performs a BER encoding of the data structure.

```
class my_sequence : public SEQUENCE {
public:
        REQUIRED<INTEGER>    i;
        OPTIONAL<INTEGER>    j;
        DEFAULT  <REAL>      x;

        my_sequence() : i (this), j (this), x (this) { x.default_val = 0.0; }
};

my_sequence   seq;

seq.i = 10;
```

```
seq.j = 20;
seq.x = 6.6;

PE<BER> pe(seq);    // encode using BER
```

The following code fragment decodes a my_sequence data structure:

```
my_sequence   seq;
PE<BER>       pe;

// get data from network

seq = pe;              // decode using BER
```

Note that a single line of code suffices for encoding oor decoding, regardless of the complexity of the data structure.

Password Lookup

This presentation concludes with a complete definition of the data structures used by the ISODE sample application, password lookup [Rose91b]. First, the ASN.1 definitions are shown, followed by the corresponding definitions using the ASN class hierarchy. While it is not difficult to translate ASN.1 specifications to the C++ classes

manually, it is clear that a translator should be constructed to perform such translations. Such work is currently in progress.

Figure A.6 shows the ASN.1 definitions for the UserName, UserID, and GroupID types used in the definition of the password lookup message. These types are then used in the definition of the Passwd message shown in Figure A.7. Figure A.8 shows the ASN class library definitions corresponding to the simple types defined in Figure A.6 for UserName, UserID, and GroupID. Finally, Figure A.9 shows the C++ definition of a Passwd data structure corresponding to the Passwd data structure defined in Figure A.7.

```
UserName ::=
    [APPLICATION 2]
        IMPLICIT GraphicString

UserID ::=
    [APPLICATION 3]
        IMPLICIT INTEGER

GroupID ::=
    [APPLICATION 4]
        IMPLICIT INTEGER
```

Figure A.6 - Password Lookup Simple Type Definitions [Rose91b]

```
Passwd ::=
    [APPLICATION 1]
        IMPLICIT SEQUENCE {
            name[0]
                IMPLICIT UserName,
            passwd[1]
                IMPLICIT IA5String
                OPTIONAL,
            uid[2]
                IMPLICIT UserID,
            gid[3]
                IMPLICIT GroupID,
            quota[4]
                IMPLICIT INTEGER
                DEFAULT 0,
            comment[5]
                IMPLICIT IA5String
                OPTIONAL,
            gecos[6]
                IMPLICIT IA5String
                OPTIONAL,
            dir[7]
                IMPLICIT IA5String
                OPTIONAL,
            shell[8]
                IMPLICIT IA5String
                OPTIONAL,
        }
```

Figure A.7 - Passwd Lookup Message [Rose91b]

```
typedef UserName    IMPLICIT<GraphicString, APPLICATION, 2>;

typedef UserID      IMPLICIT<INTEGER, APPLICATION, 3>;

typedef GroupID     IMPLICIT<INTEGER, APPLICATION, 4>;
```

Figure A.8 - Class Definitions for Password Lookup Simple Types

```
class Password : public SEQUENCE {
public:
        REQUIRED<IMPLICIT<UserName,        CONTEXT, 0> >    name;
        OPTIONAL<IMPLICIT<IA5String,       CONTEXT, 1> >    passwd;
        REQUIRED<IMPLICIT<UserID,          CONTEXT, 2> >    uid;
        REQUIRED<IMPLICIT<GroupID,         CONTEXT, 3> >    gid;
        DEFAULT  <IMPLICIT<INTEGER,        CONTEXT, 4> >    quota;
        OPTIONAL<IMPLICIT<IA5String,       CONTEXT, 5> >    comment;
        OPTIONAL<IMPLICIT<IA5String,       CONTEXT, 6> >    gecos;
        OPTIONAL<IMPLICIT<IA5String,       CONTEXT, 7> >    dir;
        OPTIONAL<IMPLICIT<IA5String,       CONTEXT, 8> >    shell;

        Password() : name(this), passwd(this), uid(this), gid(this), quota(this, 0),
                        comment(this), gecos(this), dir(this), shell(this) { }
};

typedef IMPLICIT<Password, APPLICATION, 1>  Passwd;
```

Figure A.9 - Passwd data structure

Appendix B

Source Code

```
/* ASN.h - ISO Abstract Syntax Notation abstractions */

// $Header$

/*          Copyright (c) 1993 by Virginia Polytechnic Institute
 *                          ALL RIGHTS RESERVED
 *
 *                              OOSI NOTICE
 *
 *    Acquisition, use, and distribution of this module and related
 *    materials are subject to the restrictions of a license agreement.
 *    Consult the Preface in the OOSI Manual for the full terms of
 *    this agreement.
 *
 *    Authors: Dennis Kafura, Greg Lavender, Bob Mullins.
 */

/*
 * $Log$
 */

#ifndef _OOSI_ASN_h
#define _OOSI_ASN_h

#include <BitString.h>
#include <String.h>
#include "DataUnits.h"

#define PDU_LEN 2000000
```

```
/* Map C++ primitive types onto ASN primitive types */

typedef int     AsnBool;
typedef long    AsnInt;
typedef double          AsnReal;

class AsnType;
class AsnObj;
class AsnNull;
```

```
/* ASN type tagging */

typedef octet           TagClass;
typedef octet           TagForm;
typedef unsigned short      TagId;

const TagClass UNIVERSAL        = 0x0;
const TagClass APPLICATION      = 0x1;
const TagClass CONTEXT          = 0x2;
const TagClass PRIVATE          = 0x3;

struct AsnTag {

    TagClass   cl;      // class
    TagForm    fm;      // form
    TagId      id;      // identifier

    AsnTag(TagClass c, TagForm f, TagId i) : cl(c), fm(f), id(i) {}
    void print();
};


inline
void AsnTag::print()
{
        switch (cl) {
                case UNIVERSAL:   cout << "UNIV ";
                                                break;
                case APPLICATION: cout << "APPL " << id << "\n";
                                return;
                case CONTEXT:     cout << "CNTX " << id << "\n";
                                                return;
                case PRIVATE:     cout << "PRIV " << id << "\n";
                                                return;
                default:                cout << "Unknown Class " << cl << " ";
                                                break;
        } // end switch

        switch (id) {
                case 0:   cout << "No Type Info \n";
```

```
                break;
            case 1:   cout << "BOOL\n";
                      break;
            case 2:   cout << "INT\n";
                      break;
            case 3:   cout << "BITSTR\n";
                      break;
            case 4:   cout << "OCTSTR\n";
                      break;
          case 5:   cout << "NULL\n";
                    break;
          case 6:   cout << "OBJECT IDENTIFIER\n";
                    break;
            case 9:   cout << "REAL\n";
                      break;
            case 16:  cout << "SEQ\n";
                      break;
            case 17:  cout << "SET\n";
                      break;
            case 22:  cout << "IA5STR\n";
                      break;
            case 25:  cout << "GRAPHSTR\n";
                      break;
            default:  cout << "Unknown type " << id << "\n";
                      break;
    } // end switch
}
```

```
/* A ``meta`` PE is an abstract Presentation Element. The MetaPE is
 * required so that an arbitrary ASN type can request encoding by
 * invoking one of the MetaPE virtual methods. The encode method
 * defined in the AsnPrim class is written as:
 *
 *      int encode(MetaPE* pe) { pe->prim2pe(prim, C, I); }
 *
 * This allows an arbitrary ASN object to encode its value and
 * tag since the correct `prim2pe' method will be invoked using
 * compiler generated type discrimination based on the type of
 * the `prim' argument.
 *
 * The inherited Pdu class provides the octet* vector and linkage
 * required by the protocol layers to manipulate a particular presentation
 * encoding as a general protocol data unit object.
 */

class MetaPE : public Pdu {
public:

    MetaPE() : Pdu(PDU_LEN) {}
    MetaPE( octet* data, int len) : Pdu(data, len) {}    // DK
    virtual ~MetaPE() {}

    /* Primitive type encoders/decoders */

    virtual int prim2pe(const AsnBool&, const AsnTag&) = 0;;
    virtual int pe2prim(AsnBool&, const AsnTag&) = 0;

    virtual int prim2pe(const AsnInt&, const AsnTag&) = 0;
    virtual int pe2prim(AsnInt&, const AsnTag&) = 0;

    virtual int prim2pe(const AsnReal&, const AsnTag&) = 0;
    virtual int pe2prim(AsnReal&, const AsnTag&) = 0;

    virtual int prim2pe(const String&, const AsnTag&) = 0;
    virtual int pe2prim(String&, const AsnTag&) = 0;

    virtual int prim2pe(const BitString&, const AsnTag&) = 0;
    virtual int pe2prim(BitString&, const AsnTag&) = 0;
```

```
    virtual int prim2pe(const AsnNull&, const AsnTag&) = 0;
    virtual int pe2prim(AsnNull&, const AsnTag&) = 0;

    virtual int prim2pe(const AsnObj&, const AsnTag&) = 0;
    virtual int pe2prim(AsnObj&, const AsnTag&) = 0;

    /* Constructed type encoder/decoder */

    virtual int cons2pe(const AsnTag&, int n = -1) = 0;
    virtual int seqof2pe(const AsnTag&, int n = -1) = 0;
    virtual int pe2cons(const AsnTag&) = 0;
    virtual int pe2seqof(const AsnTag&, int&) = 0;

    /* Type tag manipulation */

    virtual int tag2pe(const AsnTag&) = 0;
    virtual int pe2tag(const AsnTag&) = 0;

    virtual int cmptag(const AsnTag&) = 0;

    virtual void start() = 0;
    virtual void end() = 0;

};

/* AsnType is the abstract base class for all ASN.1 types */

class AsnType {
public:

    enum { PRIM_FORM, CONS_FORM };

    virtual ~AsnType() {}
    virtual void operator=(const AsnType& ) {}

    virtual int encode(MetaPE* pe) = 0;
    virtual int decode(MetaPE* pe) = 0;

    virtual const AsnTag& tag() = 0;

    virtual void print(int tab = 0) = 0;
```

```
    void indent(int tab) { for (int i = 0; i < tab; i++) cout << "    ";}

    static int primitive()      { return 0; }
    static int constructed()    { return 0; }

    static int form() { return primitive() ? PRIM_FORM : CONS_FORM; }

    virtual int required() const { return 0; }
    virtual int optional() const { return 0; }
    virtual int defaults() const { return 0; }
    virtual void set_default()   {}

    /* for constructed types only */

    virtual int add(AsnType* asn) { return -1; }
};
```

```
/* To implement a transfer syntax encoding, define the
 * encoder/decoder methods for the particular PE<T> instance,
 * where T is a type representing an encoding rule (see BER.{h,cc}).
 */

template<class T> class PE : public MetaPE {

    octet*      eod;    // end of data pointer

    void        adjust_pdu() {}

public:

    int  errno;         /* result of encoding */

    inline PE()                 { start(); }
    inline PE(octet* data, int len) : MetaPE(data, len)   // DK
                                { start(); }              // DK
    inline PE(AsnType& asn)     { start(); errno = asn.encode(this); }

    inline void operator=(AsnType& asn) { start(); errno = asn.encode(this); }

    int prim2pe(const AsnBool&, const AsnTag&);
    int pe2prim(AsnBool&, const AsnTag&);

    int prim2pe(const AsnInt&, const AsnTag&);
    int pe2prim(AsnInt&, const AsnTag&);

    int prim2pe(const AsnReal&, const AsnTag&);
    int pe2prim(AsnReal&, const AsnTag&);

    int prim2pe(const String&, const AsnTag&);
    int pe2prim(String&, const AsnTag&);

    int prim2pe(const BitString&, const AsnTag&);
    int pe2prim(BitString&, const AsnTag&);

    int prim2pe(const AsnNull&, const AsnTag&);
    int pe2prim(AsnNull&, const AsnTag&);
```

```
    int prim2pe(const AsnObj&, const AsnTag&);
    int pe2prim(AsnObj&, const AsnTag&);

    int cons2pe(const AsnTag&, int n = -1);
    int seqof2pe(const AsnTag&, int n = -1);
    int pe2cons(const AsnTag&);
    int pe2seqof(const AsnTag&, int&);

    int tag2pe (const AsnTag&);
    int pe2tag (const AsnTag&);

    int cmptag(const AsnTag&);

    void start() { errno = 0; eod = NULL; data = base; }
    void end();

};
```

```
/* ANY type */

class AsnAny : public AsnType {
protected:              // DK so AnyInvoke can access
   AsnType* any;    /* Any AsnType subtype */

public:

   AsnAny() {}
   void operator=(AsnType* asn) { any = asn; }
   void operator=(AsnAny& other) { any = other.any; }  //DK (for implicit)

   int encode(MetaPE* pe) { return any->encode(pe); }
   int decode(MetaPE* pe) { return any->decode(pe); }

   const AsnTag& tag() { return any->tag(); }

   int primitive()      const { return any->primitive(); }
   int constructed()    const { return any->constructed(); }

   int form() const { return any->form(); }

   int required() const { return any->required(); }
   int optional() const { return any->optional(); }
   int defaults() const { return any->defaults(); }

   int add(AsnType* asn) { return any->add(asn); }

   void print(int tab = 0) { any->print(tab);}       //DK (for AsnType)
};

#define ANY AsnAny
```

```
/* Primitive Type template
 *
 *      T ::= AsnBool | AsnInt | AsnReal
 *      C ::= tag class
 *      I ::= tag identifier
 */

template<class T, TagClass C, TagId I> class AsnPrim : public AsnType {
protected:

   T   prim;  /* primitive value */

public:

   int primitive() const { return 1; }

   const  AsnTag& tag()       { static AsnTag t(C, PRIM_FORM, I); return t; }
   static AsnTag& ChoiceTag() { static AsnTag t(C, PRIM_FORM, I); return t; }

   inline AsnPrim()           {}
   inline AsnPrim(AsnType* asn){ asn->add(this); }
   inline AsnPrim(const T& v)      { prim = v; }            //DK (added const)
   inline AsnPrim(const AsnPrim& p)  { prim = p.prim; }  //DK (added const)
   inline AsnPrim(MetaPE& pe)     { pe.start(); pe.pe2prim(prim, tag()); }

   void operator=(const T& v)              { prim = v; }
   void operator=(const AsnPrim& p)        { prim = p.prim; }
   void operator=(const AsnType& p)    { typedef AsnPrim& mytype;
                                  prim = ((mytype)p).prim;}
   inline void operator=(MetaPE& pe)       { pe.start(); pe.pe2prim(prim, tag());}

   inline operator T() const { return prim; }

   int encode(MetaPE* pe) { return pe->prim2pe(prim, tag()); }
   int decode(MetaPE* pe) { return pe->pe2prim(prim, tag()); }
   void print (int tab = 0);
};

template<class T, int c, int t>
void AsnPrim<T, c, t>::print(int tab = 0)
```

```
{
    indent(tab);
    cout << "( ";
    (tag()).print();
    indent(tab);
    cout << "    " << prim << "\n";
    indent(tab);
    cout << ")\n";
    return;
}


typedef AsnPrim<AsnBool, UNIVERSAL, 1>     BOOLEAN;
typedef AsnPrim<AsnInt,  UNIVERSAL, 2>     INTEGER;
typedef AsnPrim<AsnReal, UNIVERSAL, 9>     REAL;
typedef AsnPrim<AsnInt,  UNIVERSAL, 10>    ENUMERATED;



template<class T, TagClass C, TagId I> class AsnImplicit : public T {
public:
   const  AsnTag& tag()       { static AsnTag t(C, form(), I); return t; }
   static AsnTag& ChoiceTag() { static AsnTag t(C, form(), I); return t; }

   void operator=(const T& v)       { T::operator=(v); }
   void operator=(const AsnType& a) { T::operator=(a); }
};
#define IMPLICIT AsnImplicit


template<class T, TagClass C, TagId I> class AsnExplicit : public T {
public:

   static AsnTag& ChoiceTag()  { static AsnTag t(C, form(), I); return t; }
   inline void operator=(T& v) { T::operator=(v); }
   inline void operator=(MetaPE& pe) { decode(&pe); }
   int encode(MetaPE* pe) {
       static AsnTag t(C, CONS_FORM, I);
       int result = pe->cons2pe(t);
       if (result == 0)
           result = T::encode(pe);
```

```
        pe->end();
        return result;
    }
    int decode(MetaPE* pe) {
        static AsnTag t(C, CONS_FORM, I);
        int result = pe->pe2cons(t);
        if (result == 0)
            result = T::decode(pe);
        pe->end();
        return result;
    }
    void print(int tab = 0) {
        static AsnTag t(C, CONS_FORM, I);
        indent(tab);
        cout << "( ";
        t.print();
        T::print(++tab);
        indent(--tab);
        cout << ")\n";
    }
};
#define EXPLICIT AsnExplicit
```

```
/* BIT STRING */

template<TagClass C, TagId I> class AsnBits : public BitString, public AsnType {
typedef AsnBits mytype;
public:

    int primitive() const { return 1; }

    const AsnTag& tag() { static AsnTag t(C, PRIM_FORM, I); return t; }

    inline AsnBits() : BitString() {}
    inline AsnBits(unsigned int i)        : BitString() { BitString::set(i); }
    inline AsnBits(const BitString& b)    : BitString(b) {}
    inline AsnBits(const BitSubString& y) : BitString(y) {}
    inline AsnBits(MetaPE& pe)      { pe.start(); pe.pe2prim(*this, tag()); }

    //inline void operator=(unsigned int bit) { BitString::operator=(bit); }
    inline void operator=(unsigned int bit)   { BitString::set(bit); }
    inline void operator=(const BitString& y) { BitString::operator=(y); }
    inline void operator=(const BitSubString& y) { BitString::operator=(y); }
    inline void operator=(MetaPE& pe) { pe.start(); pe.pe2prim(*this, tag()); }
    inline void operator=(const AsnType& a)
                        { BitString::operator=((mytype)a); }

    int encode(MetaPE* pe) { return pe->prim2pe(*this, tag()); }
    int decode(MetaPE* pe) { return pe->pe2prim(*this, tag()); }
    void print(int tab = 0);
};

template<TagClass c, int t>
void AsnBits<c, t>::print(int tab = 0)
{
    indent(tab);
    cout << "( ";
    (tag()).print();
    indent(tab);
    cout << "    " << *this << "\n";
    indent(tab);
    cout << ")\n";
    return;
}
```

```
typedef AsnBits<UNIVERSAL, 3>          BIT_STRING;
```

```
/* OCTET STRINGs */

template<TagClass C, TagId I> class AsnStr : public String, public AsnType {
typedef AsnStr& mytype;
public:

    int primitive() const { return 1; }

    const  AsnTag& tag()        { static AsnTag t(C, PRIM_FORM, I); return t; }
    static AsnTag& ChoiceTag() { static AsnTag t(C, PRIM_FORM, I); return t; }

    inline AsnStr() : String() {}
    inline AsnStr(AsnType* asn){ asn->add(this); }
    inline AsnStr(const String& x) : String(x) {}
    inline AsnStr(const SubString&  x) : String(x) {}
    inline AsnStr(const char* t) : String(t) {}
    inline AsnStr(const char* t, int len) : String(t, len) {}
    inline AsnStr(char c) : String(c) {}
    inline AsnStr(MetaPE& pe)       { pe.start(); pe.pe2prim(*this, tag()); }

    inline void operator=(const AsnType& a)     {String::operator=((mytype)a);}
    inline void operator=(const String& y)  { String::operator=(y); }
    inline void operator=(const char* y)      { String::operator=(y); }
    inline void operator=(char c)               { String::operator=(c); }
    inline void operator=(const SubString&  y)      { String::operator=(y); }
    inline void operator=(MetaPE& pe) { pe.start(); pe.pe2prim(*this, tag()); }

    int encode(MetaPE* pe) { return pe->prim2pe(*this, tag()); }
    int decode(MetaPE* pe) { return pe->pe2prim(*this, tag()); }
    void print (int tab = 0);
};

template<TagClass c, int t>
void AsnStr<c, t>::print(int tab = 0)
{
     indent(tab);
     cout << "( ";
     (tag()).print();
     indent(tab);
     cout << "     " << (const char*) *this << "\n";
```

```
    indent(tab);
    cout << ")\n";
    return;
}
typedef AsnStr<UNIVERSAL, 4>   OCTET_STRING;

/* IA5String ::= [UNIVERSAL 22] IMPLICIT OCTET STRING */
typedef IMPLICIT<OCTET_STRING, UNIVERSAL, 22> IA5String;

/* NumericString ::= [UNIVERSAL 18] IMPLICIT IA5String */
typedef IMPLICIT<IA5String, UNIVERSAL, 18> NumericString;

/* PrintableString ::=  [UNIVERSAL 19] IMPLICIT IA5String */
typedef IMPLICIT<IA5String, UNIVERSAL, 19> PrintableString;

/* T61String ::= [UNIVERSAL 20] IMPLICIT OCTET STRING */
typedef IMPLICIT<OCTET_STRING, UNIVERSAL, 20> T61String;

/* TeletexString ::= T61String */
typedef T61String TeletexString;

/* VideotexString ::= [UNIVERSAL 21] IMPLICIT OCTET STRING */
typedef IMPLICIT<OCTET_STRING, UNIVERSAL, 21> VideotexString;

/* VisibleString ::= [UNIVERSAL 26] IMPLICIT OCTET STRING */
typedef IMPLICIT<OCTET_STRING, UNIVERSAL, 26> VisibleString;

/* ISO646String ::= VisibleString */
typedef VisibleString ISO646String;

/* UTCTime ::= [UNIVERSAL 23] IMPLICIT VisibleString */
typedef IMPLICIT<VisibleString, UNIVERSAL, 23> UTCTime;

/* UniversalTime ::= UTCTime */
typedef UTCTime UniversalTime;

/* GeneralizedTime ::= [UNIVERSAL 24] IMPLICIT VisibleString */
typedef IMPLICIT<VisibleString, UNIVERSAL, 24> GeneralizedTime;

/* GeneralisedTime ::= GeneralizedTime */
typedef GeneralizedTime GeneralisedTime;
```

```
/* GraphicString ::= [UNIVERSAL 25] IMPLICIT OCTET STRING */
typedef IMPLICIT<OCTET_STRING, UNIVERSAL, 25> GraphicString;

/* ObjectDescriptor ::= [UNIVERSAL 7] IMPLICIT GraphicString */
typedef IMPLICIT<GraphicString, UNIVERSAL, 7> ObjectDescriptor;

/* GeneralString ::= [UNIVERSAL 27] IMPLICIT OCTET STRING */
typedef IMPLICIT<OCTET_STRING, UNIVERSAL, 27> GeneralString;

/* CharacterString ::= [UNIVERSAL 28] IMPLICIT OCTET STRING */
typedef IMPLICIT<OCTET_STRING, UNIVERSAL, 27> CharacterString;
```

```
class AsnObj : public String, public AsnType {
typedef AsnObj mytype;

public:

    int primitive() const { return 1; }

    const AsnTag& tag() { static AsnTag t(UNIVERSAL, PRIM_FORM, 6); return t; }

    inline AsnObj() : String() {}
    inline AsnObj(AsnType* asn){ asn->add(this); }
    inline AsnObj(const String& x) : String(x) {}
    inline AsnObj(const SubString&  x) : String(x) {}
    inline AsnObj(const char* t) : String(t) {}
    inline AsnObj(const char* t, int len) : String(t, len) {}
    inline AsnObj(char c) : String(c) {}
    inline AsnObj(MetaPE& pe)       { pe.start(); pe.pe2prim(*this, tag()); }

    inline void operator=(const String& y)  { String::operator=(y); }
    inline void operator=(const char* y)    { String::operator=(y); }
    inline void operator=(char c)           { String::operator=(c); }
    inline void operator=(const SubString&  y)      { String::operator=(y); }
    inline void operator=(const AsnType& a)     { String::operator=((mytype)a);}
    inline void operator=(MetaPE& pe) { pe.start(); pe.pe2prim(*this, tag()); }

    int encode(MetaPE* pe) { return pe->prim2pe(*this, tag()); }
    int decode(MetaPE* pe) { return pe->pe2prim(*this, tag()); }

    void print(int tab = 0) {
        indent(tab);
        cout << "( ";
        (tag()).print();
        indent(tab);
        cout << "     " << (const char*) *this << "\n";
        indent(tab);
        cout << ")\n";
        return;
    }
```

```
};

typedef AsnObj OBJECT_IDENTIFIER;

class AsnNull : public AsnType {
public:
    int encode(MetaPE* pe) { return pe->prim2pe(*this, tag()); }
    int decode(MetaPE* pe) { return pe->pe2prim(*this, tag()); }
    int primitive() const { return 1; }

    const AsnTag& tag() { static AsnTag t(UNIVERSAL, PRIM_FORM, 5); return t; }

    void print(int tab = 0) {
        indent(tab);
        cout << "( ";
        (tag()).print();
        indent(tab);
        cout << ")\n";
        return;
    }
};

typedef AsnNull ASN_NULL;


/* Constructed types */

template<class T, TagClass C, int I> class AsnCons : public AsnType {
protected:

    T** cons;
    int size;
    int n;
    typedef AsnCons& mytype;
public:

    AsnCons(int s = 16) : n(-1) { cons = new T*[size = s]; }
    AsnCons(AsnType* asn, int s = 16) : n(-1) {
                                cons = new T*[size = s]; asn->add(this); }
    AsnCons(MetaPE& pe)         { pe.start(); (void) decode(&pe); }
    ~AsnCons()                  { delete cons; }
```

```
    void operator=(const AsnCons& v) {
        for (int i = 0; i <= n; i++)
            *cons[i] = *(v.cons[i]);
    }
    void operator=(const AsnType& a) { operator=((mytype)a); }
    void operator=(MetaPE& pe)       { pe.start(); (void) decode(&pe); }

    constructed() const { return TRUE; }

    const AsnTag& tag()       { static AsnTag t(C, CONS_FORM, I); return t; }
    static const AsnTag& ChoiceTag() {
        static AsnTag t(C, CONS_FORM, I); return t; }

    //int add(AsnType* asn) { cons[++n] = asn; return n + 1; }
    int add(T* asn) { cons[++n] = asn; return n + 1; }

    int encode(MetaPE* pe) {
        int result = pe->cons2pe(tag(), n + 1);
        for (int i = 0; i <= n && result == 0; i++)
            result = cons[i]->encode(pe);
        pe->end(); // pe->cons2pe(nulltag, 0);???
        return result;
    }

    void print(int tab = 0);
    //virtual int decode(MetaPE* pe) = 0;
};


template<class T, TagClass c, int t>
void AsnCons<T, c, t>::print(int tabs = 0)
{
    indent(tabs);
    cout << "( ";
    (tag()).print();
    tabs++;
    for (int i = 0; i < (n+1); i++)
        cons[i]->print(tabs);

    tabs--;
    indent(tabs);
```

```
    cout << ")\n";
    return;
}


#define CONSTRUCTED    AsnCons


class SEQUENCE : public CONSTRUCTED<AsnType, UNIVERSAL, 16> {
public:
    SEQUENCE(int sz = 16) : CONSTRUCTED<AsnType, UNIVERSAL, 16> (sz){}
    SEQUENCE(AsnType* asn) : CONSTRUCTED<AsnType, UNIVERSAL,16>
(asn) {}
    void operator= (MetaPE& pe) {
        CONSTRUCTED<AsnType, UNIVERSAL, 16>::operator= (pe);
    }
    int decode(MetaPE* pe)
    {
        int result;
        int i = 0;

        if (pe->cmptag(tag()) == TRUE)
            result = pe->pe2cons(tag());
        else
            return -1;

        while (i <= n && result == 0) {

            register AsnType* asn = cons[i++];

            if ((result = pe->cmptag(asn->tag())) == TRUE)
                result = asn->decode(pe);
            else if (asn->optional())
                continue;
            else if (asn->defaults())
                asn->set_default();
                //cout << "initialize with default\n";
            else
                return -1;
        }
```

```
        pe->end();
        return result;
    }
}; // end of SEQUENCE

template<class T>
class SEQUENCE_OF : public CONSTRUCTED<T, UNIVERSAL, 16> {
public:
    SEQUENCE_OF(int sz = 16) : CONSTRUCTED<T, UNIVERSAL, 16> (sz){
        for (int i = 0; i < sz; i++) {
            T* elem = new T;
            add(elem);
        }
    }

    T& operator[](int i) { return *cons[i]; }
    void operator= (MetaPE& pe) {
        CONSTRUCTED<T, UNIVERSAL, 16>::operator= (pe);
    }

    int encode(MetaPE* pe) {
       int result = pe->seqof2pe(tag(), n + 1);
       for (int i = 0; i <= n && result == 0; i++)
          result = cons[i]->encode(pe);
       pe->end(); // pe->cons2pe(nulltag, 0);???
       return result;
    }


    int decode(MetaPE* pe)
    {
        int result;
        int i = 0;
        int len;

        if (pe->cmptag(tag()) == TRUE)
            result = pe->pe2seqof(tag(), len); // XDR sets len to # elem
        else                                   // len < 0 - indefinent len
            return -1;

        while ((i <= n) && ((len < 0) || (i <= len)) && (result == 0) ){
```

```
            register AsnType* asn = cons[i++];

            if (pe->cmptag(asn->tag()) == TRUE)
                result = asn->decode(pe);
            else
                return -1;
        }

        pe->end();
        return result;
    }

};

class SET : public CONSTRUCTED<AsnType, UNIVERSAL, 17> {
public:
    SET(int sz = 16) : CONSTRUCTED<AsnType, UNIVERSAL, 17> (sz){}

    void operator= (MetaPE& pe) {
        CONSTRUCTED<AsnType, UNIVERSAL, 17>::operator= (pe);
    }
    int decode(MetaPE* pe)
    {
        int result = 0;
        int i = 0, start = n;

        if (pe->cmptag(tag()) == TRUE)
            result = pe->pe2cons(tag());

        while (i != start && result == 0) {

            register AsnType* asn = cons[i];
            if (pe->cmptag(asn->tag()) == TRUE) {
                result = asn->decode(pe);
                start = i;
            }

            i = ++i % (n + 1);
        }
        return 0;
    }
```

```
}; // end of SET

#define SET_OF SEQUENCE_OF
```

```
class MetaChoiceElem : public AsnType {
public:
    virtual AsnType* get_value() = 0;
    virtual void     create() = 0;
    virtual void     destroy() = 0;
    virtual void     operator=(AsnType&) {}
};

class CHOICE : public AsnType {
public:

    int type;
    int size;
    int nelem;
    MetaChoiceElem** cons;
    typedef MetaChoiceElem* ASNP;

    CHOICE(int s = 16) : type(-1),nelem(0) { cons = new ASNP[size = s]; }

    void operator=(CHOICE& c)     {
        if (type >= 0)
            cons[type]->destroy();
        type = c.type;
        cons[type]->create();
        *(cons[type]) = *(c.cons[type]);
    }
    void operator=(MetaPE& pe)    { decode (&pe); }
    int add(MetaChoiceElem* asn)  { cons[nelem++] = asn; return nelem; }
    const AsnTag& tag()           { return cons[type]->tag(); }
    void set_type(int n)          { type = n; }
    void print(int tab = 0)       { cons[type]->print(tab); }
    int encode(MetaPE* pe)        { return cons[type]->encode(pe); }
    int decode(MetaPE* pe) {
        for (type = 0; type < nelem; type++)
            if (pe->cmptag(tag()) )
                break;
        return cons[type]->decode(pe);
    }
};
```

```
template <class T, int n> class ChoiceElement : public MetaChoiceElem {
typedef ChoiceElement& mytype;

public:
    T*      value;
    CHOICE* m;

    ChoiceElement (CHOICE* asn) : value(0)
                                { m = asn; asn->add(this); }
    const AsnTag& tag()         { return T::ChoiceTag(); }
    int encode(MetaPE* pe)      { return value->encode(pe); }
    int decode(MetaPE* pe)      { if (!value)
                                  value = new T;
                                return value->decode(pe); }
    void print (int tab = 0)    { value->print(tab); }
    void create ()              { value = new T; }
    void destroy()              { delete value; }
    operator T&()               { return *value; }
    AsnType* get_value()        { return value; }
    void operator= (T& v)       { m->set_type(n);
                                if (!value)
                                    create();
                                value->operator=(v); }
    void operator=(AsnType& asn){ m->set_type(n);
                                value= new T;
                                mytype cep = (mytype)asn;
                                *value = *(cep.value); }

    void operator=(ChoiceElement& ce) {
                        m->set_type(n);
                        value = new T;
                        *value = *(ce.value);
    }

};


/*      The DISC_UNION type is implemented to handle a discriminated
 *      union as encoded by XDR.  This is the only data structure that
 *      includes type information in an XDR encoding.
 */
```

```
class DISC_UNION : public CHOICE {
public:
    void operator=(MetaPE& pe)    { decode (&pe); }
    int encode (MetaPE* pe) {
        pe->prim2pe((AsnInt) type, tag());    // encode the tag explicitly
        return cons[type]->encode(pe);
    }

    int decode (MetaPE* pe) {
        AsnTag tag(0, 0, 0);
        AsnInt i;
        pe->pe2prim (i, tag);
        type = i;
        return cons[type]->decode(pe);
    }
};
```

```
/* REQUIRED constructed type element */

template<class T> struct AsnReqElem : public T {

    virtual void operator=(const T& v)      { T::operator=(v); }
    int required() const                    { return 1; }
};
#define REQUIRED AsnReqElem

/* OPTIONAL constructed type element */

template<class T> struct AsnOptElem : public T {

    inline void operator=(T& v)     { T::operator=(v); }
    int optional() const            { return 1; }
};
#define OPTIONAL AsnOptElem

/* DEFAULT constructed type element */

template<class T> struct AsnDefElem : public T {
public:
    T def_val;

    inline void operator=(T& v)     { T::operator=(v); }
    int defaults() const            { return 1; }
    void set_default()              { T::operator=(def_val); }
};
#define DEFAULT   AsnDefElem

#endif
```

```
#ifndef _rpc_msg_h
#define _rpc_msg_h

#include "ASN.h"

enum msg_type {
        CALL = 0,
        REPLY = 1
};

enum reply_stat {
        MSG_ACCEPTED    = 0,
        MSG_DENIED      = 1
};

enum accept_stat {
        RPC_SUCCESS             = 0,
        PROG_UNAVAIL            = 1,
        PROG_MISMATCH           = 2,
        PROC_UNAVAIL            = 3,
        GARBAGE_ARGS                    = 4
};

enum reject_stat {
        RPC_MISMATCH    = 0,
        AUTH_ERROR      = 1
};

enum auth_stat {
        AUTH_BADCRED            = 1,
        AUTH_REJECTEDCRED       = 2,
        AUTH_BADVERF            = 3,
        AUTH_REJECTEDVERF       = 4,
        AUTH_TOOWEAK            = 5
};


// currently only supports NULL Authorization


class my_auth : public SEQUENCE {
```

```
public:
        REQUIRED<INTEGER>       a;
        REQUIRED<INTEGER>       b;

        void operator=(MetaPE& pe) { SEQUENCE::operator=(pe); }
        my_auth() : a(this), b(this) {}
};


class mismatch_info : public SEQUENCE {
public:
        REQUIRED<INTEGER>   low;
        REQUIRED<INTEGER>   high;

        void operator=(MetaPE& pe) { SEQUENCE::operator=(pe); }
        mismatch_info() : low(this), high(this) {}
};


class rejected_reply : public DISC_UNION {
public:
    ChoiceElement<mismatch_info, RPC_MISMATCH>  mmi;
    ChoiceElement<INTEGER, AUTH_ERROR>          stat;

    void operator= (MetaPE& pe) { DISC_UNION::operator=(pe); }
    void operator= (rejected_reply& v) {}
    rejected_reply() : mmi(this), stat(this) {}
};


template <class T> class reply_data_type : public DISC_UNION {
public:
    ChoiceElement< T, RPC_SUCCESS>              results;
    ChoiceElement<mismatch_info, PROG_MISMATCH> mmi;

    void operator= (MetaPE& pe) { DISC_UNION::operator=(pe); }
    void operator= (reply_data_type& v) {}
    reply_data_type() : results(this), mmi(this) {}
};

template <class T> class accepted_reply : public SEQUENCE {
```

```
public:
    REQUIRED<my_auth>              verf;
    REQUIRED<reply_data_type< T > > reply_data;

    void operator=(MetaPE& pe) { SEQUENCE::operator=(pe); }
    void operator=(accepted_reply& v) {}
    accepted_reply() : verf(this), reply_data(this) {}
};



template <class T> class reply_body : public DISC_UNION {
public:
    ChoiceElement<accepted_reply< T >, MSG_ACCEPTED>  areply;
    ChoiceElement<rejected_reply, MSG_DENIED>         rreply;

    void operator= (MetaPE& pe) { DISC_UNION::operator=(pe); }
    void operator= (reply_body& v) {}
    reply_body() : areply(this), rreply(this) {};
};


template <class T> class call_body : public SEQUENCE {
public:
        REQUIRED<INTEGER>    rpcvers;
        REQUIRED<INTEGER>    prog;
        REQUIRED<INTEGER>    vers;
        REQUIRED<INTEGER>    proc;
        REQUIRED<my_auth>    cred;
        REQUIRED<my_auth>    verf;
        REQUIRED<T >         Args;     // procedure specific stuff


       void operator=(MetaPE& pe) { SEQUENCE::operator=(pe); }
       void operator=(call_body& v) {}
       call_body() : rpcvers(this), prog(this), vers(this), proc(this),
                cred(this), verf(this), Args(this) {}
};

template <class T> class body_type : public DISC_UNION {
public:
```

```
    ChoiceElement<call_body < T >, CALL>   cbody;
    ChoiceElement<reply_body< T >, REPLY>  rbody;

    void operator= (MetaPE& pe) { DISC_UNION::operator=(pe); }
    void operator= (body_type& v) {}
    body_type() : cbody(this), rbody(this) {};
};

template <class T> class rpc_msg : public SEQUENCE {
public:
    REQUIRED<INTEGER>          xid;
    REQUIRED<body_type< T > >  body;

    void operator=(MetaPE& pe) { SEQUENCE::operator=(pe); }
    rpc_msg() : xid(this), body(this) {}
};

class mapping : public SEQUENCE{
public:
        REQUIRED<INTEGER> prog;
        REQUIRED<INTEGER> vers;
        REQUIRED<INTEGER> prot;
        REQUIRED<INTEGER> port;

       void operator=(MetaPE& pe) { SEQUENCE::operator=(pe); }
        mapping() : prog(this), vers(this), prot(this), port(this){}
};



#endif
```

```
#ifndef _mysocket_h
#define _mysocket_h

extern "C" {
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <unistd.h>

#define INADDR_NONE -1

extern unsigned long inet_addr(char* s);

}

#define SERVER_PORT     6543
#define SERVER_HOST     "128.173.6.72"

enum PROCESS_TYPE { INITIATOR, RESPONDER };

class socket_base {
protected:
        int                     sockfd, newsockfd;
        struct sockaddr_in      cli_addr, serv_addr;
        PROCESS_TYPE            proc_type;

        int get_addr (char*);

public:
        int a_request(char* protocol, char* serv_name, int port);
        int a_init(char* protocol, int port);
        int a_response();
        int r_request() {close(newsockfd); return (close(sockfd)); }
};

int socket_base::a_request (char* protocol, char* serv_name, int port)
```

```
{
        proc_type = INITIATOR;

        if ((sockfd = socket (AF_INET, SOCK_DGRAM, 0)) < 0) {
                perror ("server:  cannot create socket");
                exit (1);
        }

        memset ((char *) &serv_addr, 0, sizeof (serv_addr));
        serv_addr.sin_family            = AF_INET;
        serv_addr.sin_addr.s_addr       = get_addr(serv_name);
        serv_addr.sin_port                  = htons (port);

        memset ((char*) &cli_addr, 0, sizeof(cli_addr));
        cli_addr.sin_family             = AF_INET;
        cli_addr.sin_addr.s_addr        = htonl(INADDR_ANY);
        cli_addr.sin_port               = htons(0);

        if (bind(sockfd, (struct sockaddr *)&cli_addr, sizeof(cli_addr))
                                                                < 0) {
                perror("a_request: can't bind local address");
                exit(1);
        }
        return 1;
}


int socket_base::a_init (char* protocol, int port)
{
        proc_type = RESPONDER;
        if ((sockfd = socket (AF_INET, SOCK_DGRAM, 0)) < 0) {
                perror ("server:  cannot create socket");
                exit (1);
        }

        memset ((char *) &serv_addr, 0, sizeof (serv_addr));
        serv_addr.sin_family            = AF_INET;
        serv_addr.sin_addr.s_addr       = htonl (INADDR_ANY);
        serv_addr.sin_port              = htons (port);

        memset ((char *) &cli_addr, 0, sizeof (cli_addr));
```

```
        cli_addr.sin_family             = AF_INET;
        cli_addr.sin_addr.s_addr        = htonl (INADDR_ANY);

        if (bind (sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr))
                                                                < 0){
                perror ("server bind error");
                exit (1);
        }

        listen (sockfd, 5);
        return 1;
}


int socket_base::a_response()
{

        int clilen = sizeof (cli_addr);
        newsockfd = accept (sockfd, (struct sockaddr *) &cli_addr, &clilen);

        if (newsockfd < 0) {
                perror ("server:  accept error");
                exit (1);
        }
        return newsockfd;
}

int socket_base::get_addr (char* host)
{
        int                     ip_addr;
        struct hostent  *hp;
        struct in_addr *ptr;


        /*  First try to convert the host name from dotted decimal.  If that
         *  fails, call gethostbyname
         */

        if ((ip_addr = inet_addr (host)) != INADDR_NONE)
                return ip_addr;
        else {
```

```
                if ((hp = gethostbyname (host)) == NULL) {
                        perror ("unknown host name");
                        exit (1);
                }
                ptr = (struct in_addr *) hp->h_addr_list[0];
                return ptr->s_addr;
        }
}



class udp_socket : public assoc_base {
public:
        int Read (unsigned char*, int);
        int Write(unsigned char*, int);
};

int udp_socket::Read (unsigned char* s, int max_len)
{
        int len, addr_len;

        addr_len = sizeof (*(struct sockaddr*)&serv_addr);
        if (proc_type == INITIATOR)
                len = recvfrom(sockfd, s, max_len, 0,
                                (struct sockaddr*)&serv_addr, &addr_len);
        else
                len = recvfrom(sockfd, s, max_len, 0,
                                (struct sockaddr*)&cli_addr,  &addr_len);

        if (len < 0)
                perror("recvfrom error");

        return len;
}

int udp_socket::Write (unsigned char* s, int max_len)
{
        int stat;

    if (proc_type == INITIATOR)
            stat = sendto(sockfd, s, max_len, 0, (char*)&serv_addr,
```

```
                                sizeof(serv_addr));
        else
                stat = sendto(sockfd, s, max_len, 0, (char*)&cli_addr,
                                        sizeof(cli_addr));
        if (stat != max_len)
                perror("sendto error");

        return stat;
}


#endif
```

```
#include <iostream.h>
#include "assoc.h"
#include "ASN.h"
#include "XDR.h"
#include "RpcMsg.h"


template <class T> class ROP {
public:
        INTEGER prog, vers, proc;
        ROP (INTEGER prg, INTEGER v, INTEGER prc) :
                        prog(prg), vers(v), proc(prc) {}
        int call_rop (char* rhost, OCTET_STRING s)
        {
                rpc_msg<T> msg;
                rpc_msg<INTEGER>        reply;
                int port = get_port(rhost, prog, vers);

                msg.xid = 1;
                msg.body.type = 0;
                msg.body.cbody.rpcvers = 2;
                msg.body.cbody.prog    = prog;
                msg.body.cbody.vers    = vers;
                msg.body.cbody.proc    = proc;
                msg.body.cbody.cred.a  = 0;
                msg.body.cbody.cred.b  = 0;
                msg.body.cbody.verf.a  = 0;
                msg.body.cbody.verf.b  = 0;
                msg.body.cbody.Args    = s;

                PE<XDR> pe(msg);

                udp_socket remote;
                remote.a_request ("udp", rhost, port);
                remote.Write (pe.base, pe.data - pe.base);
                remote.Read  (pe.base, 132);
                pe.data = pe.base;

                reply.decode(&pe);

                return reply.body.rbody.areply.reply_data.results;
```

```
        }
        int get_port (char* rhost, INTEGER prog, INTEGER vers)
        {
                rpc_msg<mapping> msg;
                rpc_msg<INTEGER> reply;

                mapping         map;

                map.prog = prog;
                map.vers = vers;
                map.prot = 17;
                map.port = 0;

                msg.xid = 1;
                msg.body.type = 0;
                msg.body.cbody.rpcvers = 2;
                msg.body.cbody.prog    = 100000;
                msg.body.cbody.vers    = 2;
                msg.body.cbody.proc    = 3;
                msg.body.cbody.cred.a  = 0;
                msg.body.cbody.cred.b  = 0;
                msg.body.cbody.verf.a  = 0;
                msg.body.cbody.verf.b  = 0;
                msg.body.cbody.Args    = map;


                PE<XDR> pe(msg);

                udp_socket remote;
                remote.a_request ("udp", rhost, 111);
                remote.Write (pe.base, pe.data - pe.base);
                remote.Read  (pe.base, 132);
                pe.data = pe.base;

                reply.decode(&pe);

                return reply.body.rbody.areply.reply_data.results;
        }
}; // end of ROP
```

```
main()
{
        INTEGER prog = 99;
        INTEGER vers = 1;
        INTEGER proc = 1;

        OCTET_STRING s = "Bob Mullins";
        ROP<OCTET_STRING> rop(prog, vers, proc);
        int stat = rop.call_rop ("actor", s);

}
```

Vita

Robert Mullins was born February 10, 1964, in Lancaster, Pennsylvania. He attended Millersville University of Pennsylvania from 1982 to 1986, earning a Bachelor of Science degree in computer science and graduating Cum Laude. After graduation from Millersville University, Robert worked for two years as a software engineer in for a hospital information system software development company. In 1988, he moved to Atlanta, Georgia to take a position as a software engineer in the network development group of another software development corporation, and served as the company's representative to the Health Level Seven (HL7) working group which was attempting to develop a network standard for electronic information interchange in the healthcare industry. In 1991, Robert started graduate school at Virginia Polytechnic Institute and State University where he worked under the direction of Dennis Kafura, graduating in 1993. Robert now works as a consultant and contract programmer, developing network solutions for various companies.