Programming with ASN.1 using Polymorphic Types and Type Specialization*

R. G. Lavender,[a] D. G. Kafura[b] and R. W. Mullins[b†]

[a]ISODE Consortium
The Dome, The Square, Richmond TW9 1DT England

[b]Department of Computer Science
562 McBryde Hall, Virginia Tech, Blacksburg, Virginia 24061 USA

A set of object-oriented abstractions is described that permits direct programming with ASN.1 specified types. The design and implementation also supports the flexible use of different encoding rules. The implementation is in C++ and makes use of *class templates* for representing polymorphic types, *class inheritance* for type specialization, and *typedefs* for defining ASN.1 types as instances of class templates. Encoding/decoding performance data is provided as evidence that this work is suitable for serious application development.

Keyword Codes: C.2.2; D.1.5; D.2.2
Keywords: Network Protocols; Object-Oriented Programming; Tools and Techniques

1. INTRODUCTION

Programming OSI applications requires that both user data and application protocols be specified in ASN.1 [8]. Current OSI programming practice relies on tools for generating language specific constructs that represent the ASN.1 types that define application data and protocols. The typical model for using ASN.1 types in an application is to automatically generate corresponding high-level language data structures representing the specification, and operations for encoding and decoding each data structure using a suitable transfer syntax, such as the Basic Encoding Rules (BER) [9]. The automatic translation from an ASN.1 specification to equivalent data structures in a target language, such as C, often requires that the programmer understand the mapping of names from the ASN.1 specification to the language specific data structures. This knowledge is required to facilitate development and debugging since automatically generated names are often "mangled" and corresponding canonical data structures are often generated for the convenience of encoder and decoder functions rather than the programmer. An application developer is left with the sometimes awkward task of manipulating the canonical data

*Published in *Proceedings of the IFIP TC6/WG6.5 International Conference on Upper Layer Protocols, Architectures and Applications*, M. Medina and N. Borenstein (editors), Barcelona, Spain, June 1994.
†D. G. Kafura and R. W. Mullins were supported in part by grant CCR-9104013 from the US National Science Foundation.

structures and invoking the encoding and decoding operations as part of exchanging data with a remote peer.

Furthermore, many ASN.1 tools and presentation service access points are not designed and implemented to support the use of multiple sets of encoding rules; instead, only a single set of encoding rules, namely the BER, are supported. Our framework is explicitly architected to encompass the current standardized encoding rules, as well as customized and de facto encoding rules; for example, the lightweight encoding rules proposed by Huitema and Doghri [6,7] and, with modest limitations, XDR [12].

Our methodology for building ASN.1-based applications is novel in two respects:

- a set of extensible object-oriented abstractions is defined that allows the natural expression and use of the *primitive*, *constructed*, and *component* ASN.1 types directly in C++.
- a generic *presentation element* (PE) abstraction is defined that facilitates the implementation of diverse encoding rules through type template instantiation.

To emphasize the first point, Figure 1 presents a simple ASN.1 module specification for a set of types that might reasonably be used to implement an application protocol for looking up a password from a Unix-like password file. Figure 1 presents C++ type definitions and a class that are automatically generated by an ASN.1 to C++ translator from the module specification in Figure 1. As illustrated, the translation preserves much of the structure and syntax of the original ASN.1 specification. The structural and syntactic coherence of the mapping from ASN.1 to C++ is achieved using a set of *polymorphic types* to construct C++ types that correspond to type definitions in ASN.1.[3]

In the remainder of this paper, we describe how such polymorphic types are constructed in C++ using *type templates* and inheritance among types instantiated from templates. Section 2 provides a brief introduction to the basic type definition and type specialization constructs of C++, while simultaneously introducing primitive type abstractions used subsequently. Section 3 defines the mapping of ASN.1 types to equivalent type abstractions in C++. Section 4 defines the presentation element abstraction and provides a comparison of encoding/decoding times among Sun XDR, C++/XDR, C++/BER, and ISODE/BER. This data demonstrates that our implementation is suitable for serious application development. Section 5 offers some observations on our research experiences in this area and compares this work to similar work by others.

2. CLASSES, TEMPLATES, AND INHERITANCE

A language supporting the definition of user-defined polymorphic types and specialization of types via inheritance is sufficiently powerful to allow the degree of type extension required to represent the primitive, constructed, and component types of ASN.1. The type definition mechanisms required include the *class*, the *template*, and the *typedef*. A class permits the definition of an abstract data type. Templates permit the definition of parameterized abstract data types. A typedef is used to define a type alias. In this section, we briefly illustrate the use of classes, templates, and inheritance in C++.

[3]An ASN.1 module is represented here using the proposed ISO/ANSI C++ *namespace* construct, which is not yet implemented by most commercial C++ language processors.

```
PasswordLookup DEFINITIONS ::=

BEGIN -- similar to an entry in <pwd.h>

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
    }

UserName ::= [APPLICATION 2] IMPLICIT GraphicString
UserID   ::= [APPLICATION 3] IMPLICIT INTEGER
GroupID  ::= [APPLICATION 4] IMPLICIT INTEGER

END
```

```
/* automatically generated -*- C++ -*- by CATY version 1.0 */
#include "UNIV.h"

namespace PasswordLookup {

    using namespace Universal;  // universal types (i.e., INTEGER, IA5String, etc.)

    typedef IMPLICIT <APPLICATION, 2, GraphicString> UserName;
    typedef IMPLICIT <APPLICATION, 3, INTEGER>       UserID;
    typedef IMPLICIT <APPLICATION, 4, INTEGER>       GroupID;

    class Passwd : public IMPLICIT <APPLICATION, 1, SEQUENCE> {
    public:

        REQUIRED <IMPLICIT <CONTEXT, 0, UserName> >  name;
        OPTIONAL <IMPLICIT <CONTEXT, 1, IA5String> > passwd;
        REQUIRED <IMPLICIT <CONTEXT, 2, UserID> >    uid;
        REQUIRED <IMPLICIT <CONTEXT, 3, GroupID> >   gid;
        DEFAULTS <IMPLICIT <CONTEXT, 4, INTEGER> >   quota;
        OPTIONAL <IMPLICIT <CONTEXT, 5, IA5String> > comment;
        OPTIONAL <IMPLICIT <CONTEXT, 6, IA5String> > gecos;
        OPTIONAL <IMPLICIT <CONTEXT, 7, IA5String> > dir;
        OPTIONAL <IMPLICIT <CONTEXT, 8, IA5String> > shell;

        Passwd() : name(this), passwd(this), uid(this),
        gid(this), quota(this), comment(this), gecos (this), dir(this),
        shell(this)    { quota.def_val = 0; }
    };
}
```

Figure 1. Password Lookup ASN.1 Specification and Corresponding C++.

```
class AsnType {
public:          /* methods visible to clients and derived classes */

   virtual ~AsnType() {}          /* empty destructor */

   /* default methods to be selectively redefined by a derived class */

   virtual bool primitive()       const { return FALSE; }
   virtual bool constructed()     const { return FALSE; }

   virtual bool required()        const { return FALSE; }
   virtual bool optional()        const { return FALSE; }
   virtual bool defaults()        const { return FALSE; }

   /* "pure" methods to be defined by a derived class */

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

   virtual const Tag& tag() = 0;
};
```

Figure 2. An "abstract" ASN.1 base class.

Both the class and the template permit the definition of arbitrary user-defined types that effectively extend the base set of types offered in a language environment. User-defined types are further extended using either single or multiple inheritance between previously defined classes and templates.

2.1. Classes

A class in C++ defines an abstract data type consisting of data and/or methods for manipulating that data. Methods defined `virtual` may be redefined by a subclass. A special sort of virtual method is the "pure" virtual method, which has no implementation in the class in which it is first declared. A pure virtual method defines a method's type signature, but the binding of an implementation is deferred.

Figure 2 illustrates an abstract class, `AsnType`, that defines a set of virtual methods used in the implementation of ASN.1 in C++. The purpose of the `AsnType` class is to define the type signatures and default implementations of methods that are selectively overridden by a class derived from `AsnType`. The `encode`, `decode`, and `tag` methods are defined as pure virtual, meaning that a subclass must provide a default implementation. The `MetaPE` argument provided to the `encode/decode` method denotes an abstract presentation element that "knows" how to encode/decode an `AsnType` to/from a specific transfer syntax.

2.2. Templates

The template definition in Figure 3 introduces a type template `Prim` parameterized by a type `T`. The purpose of the `Prim<T>` template is to allow the definition and use

```
template<class T> class Prim {
protected:                  //  visible to subclasses only

   T    val;                //  a value of type T

public:

   inline Prim()            { val = 0; }    //  default constructors
   inline Prim(T v)         { val = v; }

   inline void operator=(T v) { val = v; }          //  overloaded assignment
   inline operator T() const { return val; }   //  type converter
};

typedef Prim<bool> Boolean;
```

Figure 3. A template for primitive types.

of primitive language types as first-class objects. In many object-oriented languages, primitive types (e.g., `int`), are not implemented as first-class objects so that efficient machine code can be generated. The `Prim<T>` template is purposefully constructed so as to incur no extra space overhead in representing a primitive type and all methods are defined inline to eliminate procedure call overhead when invoking operations at run-time; hence, the efficiency the `Prim<T>` abstraction is close to optimal. A type conversion operator, `operator T`, is defined so that a `Prim<T>` object may be implicitly converted to a value of type `T`. For example, the type `Boolean` is defined as an instance of the `Prim<T>` template by supplying the appropriate type parameter; in this case, the enumerated type `bool`. The instantiation of a `Boolean` type results in the instantiation of a `Prim<bool>` instance, which can then be used in an expression like a `bool` value.

2.3. Inheritance

Figure 4 illustrates the use of inheritance in defining a template `ArithPrim<T>` as a specialization of the `Prim<T>` class presented in Figure 3. The `ArithPrim<T>` template inherits publicly from `Prim<T>`, meaning that a combined public interface is visible to clients of an instance of `ArithPrim<T>`. Note that the type parameter `T` is propagated through inheritance to the `Prim<T>` base class. The composite object consists of a private instance variable `val` of type `T` and a set of public inline methods implementing primitive arithmetic operations on that value. The definition of `Integer` and `Real` types results in the instantiation of primitive arithmetic types based on a `long` and a `double`, respectively.[4] The run-time efficiency cost for representing first-class `Integer` and `Real` objects in this manner is minimal. The advantage is that additional types can be defined that are incremental extensions of these primitive arithmetic objects.

[4]Alternatively, a `long long` and `long double` could be used to extend the range of the value sets.

```
template<class T> class ArithPrim : public Prim<T> {
public:

    inline void operator=(T v)  { Prim<T>::operator=(v); }

    inline T operator += (T v)     { return val += v; }
    inline T operator −= (T v)     { return val −= v; }
    inline T operator *= (T v)     { return val *= v; }
    inline T operator /= (T v)     { return val /= v; }
    inline T operator ++ ()        { return ++val; }
    inline T operator −− ()        { return −−val; }
};

typedef ArithPrim<long>        Integer;
typedef ArithPrim<double>      Real;
```

Figure 4. The `ArithPrim` class derived from `Prim<T>`.

3. MAPPING ASN.1 TO TYPE TEMPLATES

Our goal in defining a mapping of ASN.1 into C++ is to allow the application layer programmer to define and use ASN.1 type instances as objects, with all the attendant benefits of an object-oriented representation of application data. A critical step towards this goal is the definition of a set of *type templates* that can either be automatically generated from a formal ASN.1 specification, or used directly in an ad hoc manner by a programmer constructing a custom application. From these type templates, we should be able to instantiate a set of C++ types, representing an ASN.1 module, that can be used naturally by a programmer building a distributed application. The present state of the art relies almost exclusively on programming using the canonical data structures output from an ASN.1 translator, rather than the data representations originally specified in ASN.1. Our objective is to minimize the gap between ASN.1 as a specification language and the type system of a candidate implementation language. The high-level expressiveness of the ASN.1 language typically exceeds that of the type definition facilities of the typical programming language used to construct an OSI application. If we can approach the same level of type expressiveness of ASN.1 *within* the confines of the type system of an object-oriented language, namely C++, then we have made a significant step towards facilitating application development.

ASN.1 types come in three flavors: primitive types, constructed types, and component types. For example, the ASN.1 type names **BOOLEAN**, **INTEGER**, **REAL**, and **OCTET STRING** define primitive types; **SEQUENCE** and **SET** define constructed types; and **ANY** and **CHOICE** define component types. A mapping of ASN.1 types to a particular target language involves defining an appropriate data structure for each type, and taking into consideration the fact that types in ASN.1 are augmented with tag information that is used to uniquely identify the type of a value that is to be encoded in a machine independent manner.

In defining a translation from ASN.1 to C++, we conceptually map each ASN.1 type

to a *tagged type tuple* of the form:

<tag-class, tag-id, base-type>

The tag information is defined by two components: a *tag-class* and a *tag-id*. A tag class is one of **UNIVERSAL**, **APPLICATION**, **PRIVATE**, or **CONTEXT** (context-specific), and a tag-id is an arbitrary non-negative integer value. We think of a tagged type tuple as denoting an extended type in the target language. The element *base-type* corresponds to either a primitive type in the target language (e.g., the `Integer` and `Real` primitive types defined previously), or a type constructed from the primitive types. Since our target language supports a type definition facility, we ought to be able to define some subset of the ASN.1 types using a type definition of the form:

`typedef` *base-type type-alias*;

A `typedef` introduces a new type name as an alias for an existing type; for example, to accommodate anglophiles:

```
typedef GeneralizedTime GeneralisedTime;
```

A type definition may also be used to instantiate a type template corresponding to a type defined in an ASN.1 specification, as was done previously in defining the `Boolean`, `Integer`, and `Real` types. This form of type definition requires a template name, and some number of parameters; for example, a tag class, a tag id, and a base type as follows:

`typedef` *template-name<tag-class, tag-id, base-type> type-alias*;

We wish to define all of the primitive ASN.1 types as type tuples of this form and then instantiate them so that a programmer can use these "embedded" ASN.1 types to define application data and protocols.

3.1. Primitive Types

The `PrimType` template depicted in Figure 5 is used to define an ASN.1 primitive type that inherits from a "splitter" class `AsnPrim`. `AsnPrim` allows a `PrimType` instance to have a split personality composed from the primitive base type `T` and the abstract ASN.1 base class `AsnType` (see Figure 2). The default implementation of the virtual `primitive` method defined in the `AsnType` class is overridden, while the pure virtual `encode`, `decode`, and `tag` methods are given implementations. The `encode` and `decode` methods simply call upon an abstract presentation element (`MetaPE`) object passed as a parameter to effect the encoding and decoding of a primitive value. The details of the `MetaPE` methods are omitted; essentially, a MetaPE maps an ASN.1 object into a suitable transfer syntax according to a desired set of encoding rules. This structure ensures separation of the representation of an ASN.1 type from any encoded form, thereby achieving a high degree of independence.

The instantiation of ASN.1 primitive types belonging to the **UNIVERSAL** class is accomplished by using a `typedef` to define new type instances within a `Universal` namespace:

```
template<class T> class AsnPrim : public T, public AsnType { ... };

template<int C, int I, class T> class PrimType : public AsnPrim<T> {
public:
                                                                        5
    ... /*constructors and overloaded assignment operators */

    bool primitive() const { return TRUE; }

    /* encode/decode "self" into/from a Presentation Element (PE) */      10

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

    /* a PE<BER> encoder/decoder will want to know our tag */             15

    const Tag& tag() { static Tag t(C, PRIM, I); return t; }
};
```

Figure 5. The `PrimType` template for defining primitive ASN.1 types.

```
namespace Universal {
    typedef PrimType<UNIVERSAL, 1, Boolean>     BOOLEAN;
    typedef PrimType<UNIVERSAL, 2, Integer>     INTEGER;
    typedef PrimType<UNIVERSAL, 3, BitString>   BIT_STRING;
    typedef PrimType<UNIVERSAL, 4, OctetString> OCTET_STRING;
    ...
}
```

Each type is instantiated using a tagged type tuple as the parameter list to the `PrimType` template. The `PrimType` template is defined so that the tag-class and tag-id are used to construct a statically determined internal tag object, while the base type is propagated upward in a public inheritance hierarchy so that an instance of each newly defined type will inherit the correct behavior (e.g., arithemtic operations for an INTEGER type). Each `PrimType` instance is distinguished from the base type on which it is based by the tag information. A type defined in this manner may subsequently be used as the base type for some other type definition. For example, the **IA5String** universal type is defined in ASN.1 in terms of the **OCTET STRING** type:

IA5String ::= [UNIVERSAL 22] IMPLICIT OCTET STRING

The ASN.1 syntax is mapped into a corresponding type definition using the previous definition of the `OCTET_STRING` type:

```
typedef IMPLICIT<UNIVERSAL, 22, OCTET_STRING> IA5String;
```

The ASN.1 tag modifier **IMPLICIT** is mapped onto a template name `IMPLICIT` that ensures that the `IA5String` type is tagged with the overriding tag information, which,

```
template<class T> class SeqOfType : public ConsType<UNIVERSAL, 16, T> {
public:
    ...
    int encode(MetaPE* pe) { return pe->seqof2pe(*this, n); }
    int decode(MetaPE* pe) { return pe->pe2seqof(*this, n); }
};

#define SEQUENCE      SeqOfType<AsnType>
#define SEQUENCE_OF   SeqOfType

template<class T> class SetOfType : public ConsType<UNIVERSAL, 17, T> {
public:
    ...
    int encode(MetaPE* pe) { return pe->setof2pe(*this, n); }
    int decode(MetaPE* pe) { return pe->pe2setof(*this, n); }
};

#define SET           SetOfType<AsnType>
#define SET_OF        SetOfType
```

Figure 6. Templates for SEQUENCE and SET.

depending on the encoding rules used, may be necessary during transformation to a suitable transfer syntax.

3.2. Constructed Types

The ASN.1 constructed types **SEQUENCE** and **SET** are structurally similar. The **SEQUENCE** and **SEQUENCE OF** types share the same tag class and tag id, as do **SET** and **SET OF**. In C++, sequences and sets are represented using two distinct templates: `SeqOfType<T>` and `SetOfType<T>`. Both inherit a list-oriented structure and list manipulation operations from a common ASN.1 constructed type template `ConsType`, which is parameterized with the tagged type tuple used to define a `SeqOfType<T>` or `SetOfType<T>` instance. The details of the `ConsType` class are omitted since list representation and manipulation techniques are well known. Preprocessor macros are defined to map the ASN.1 syntax onto the C++ type templates. The abstract class `AsnType` is used for the type parameter to the `SeqOfType<T>` and `SetOfType<T>` templates so that a `SEQUENCE` or `SET` type may be composed of a list of arbitrary ASN.1 C++ types, all of which are subtypes of `AsnType` by virtue of inheritance.

Both the `SeqOfType<T>` and the `SetOfType<T>` templates define specific `encode` and `decode` operations since both are ultimately derived from `AsnType`, which requires that a derived class provide such implementations. The `tag` method is inherited from the `ConsType` class, which is passed the tag class and tag id information as part of the tagged type tuple parameter list. The `encode` method uses its presentation element argument to call a specific sequence encoder method, passing in a reference to itself (`*this`) and the number of elements in the sequence. The presentation element will then request tag information by calling the inherited `tag` method and then incrementally encoding each of

```
template<class T> class AnyType : public AsnAny {
protected:

    T*    any;      /* any ASN.1 type */

public:
    ...
    int primitive()          { return any->primitive(); }
    int constructed()        { return any->constructed(); }

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

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

    const Tag& tag() { return any->tag(); }
};
#define ANY    AnyType
```

Figure 7. The ANY type.

the elements in the sequence. The `decoding` method performs the reverse operation. A similar situation occurs when encoding and decoding sets since sets and sequences share a common representation. However, the fact that sets are unordered dictates that a slightly different encode/decode algorithm be used.

3.3. Component Types

The ASN.1 **CHOICE** and **ANY** are "containers" that do not themselves represent type information, but rather contain one or more type instances as components. The **ANY** type is represented by a container template, `AnyType<T>`, which can be instantiated to refer to any defined type `T`. As shown in Figure 7, the `AnyType<T>` template defines a method interface that maps onto the corresponding methods of the type `T` instance referenced by the identifier `any`. Since every ASN.1 type `T` is a subtype of the abstract `AsnType` class, all methods will be defined by the object referenced by `any`.

A **CHOICE** type may be viewed as a list of `AnyType<T>` instances, only one of which is valid. A **CHOICE** is defined in C++ by a `ChoiceType`, which is a container class that operates much like the `AnyType<T>` class. Rather than presenting the details of the `ChoiceType` class, we present in Figure 8 the definition of the **EXTERNAL** type, which illustrates the use of both **CHOICE** and **ANY**.

The `encoding` element of the `EXTERNAL` type is derived from the `CHOICE` type, which is simply an alias for the `ChoiceType`. An `ALTERNATE` template is used to define each element of a **CHOICE**. Each choice type also has an associated enumerated type that is used to identify which alternative element is valid.

```
struct EXTERNAL : public IMPLICIT<UNIVERSAL, 8, SEQUENCE> {

    OPTIONAL <OBJECT_IDENTIFIER> direct_reference;
    OPTIONAL <INTEGER>           indirect_reference;
    OPTIONAL <ObjectDescriptor> data_value_descriptor;

    struct encoding : public CHOICE {
        ALTERNATE <0, EXPLICIT <CONTEXT, 0, ANY> > single_ASN1_type;
        ALTERNATE <1, IMPLICIT <CONTEXT, 1, OCTET_STRING> > octet_aligned;
        ALTERNATE <2, IMPLICIT <CONTEXT, 2, BIT_STRING> > arbitrary;

        enum { typeof_single_ASN1_type, typeof_octet_aligned, typeof_arbitrary } type;
    };
};
```

Figure 8. The EXTERNAL type.

3.4. Recursive Types

Recursive type definitions present a small difficulty when defining C++ types representing a **SEQUENCE**, **SET**, or **CHOICE**. The problem is nicely handled by defining elements of a **SEQUENCE** or **SET** as either an instance of a `REQUIRED`, `OPTIONAL`, or `DEFAULTS` template. For example, the `Passwd` type defined previously in Figure 1 as a subtype of the `SEQUENCE` type has its elements defined using one or more instances of these templates. Similarly, the `ALTERNATE` template is used to define the elements of the **CHOICE** in Figure 8.

The purpose of defining elements of a constructed or component type with one of these templates is two-fold: first, the element template adds additional semantic information to the type that indicates how it is to be dealt should a value not be present at the time of encoding/decoding; secondly, a level of indirection is introduced that is compatible with the definition of recursive types in C++. The definition of a recursive type in C++ requires the use of a pointer type. The element templates each define a type that when instantiated acts as a *smart pointer,* which is a common abstraction technique used in C++ to encapsulate a pointer type so that it can be treated as a first-class object. The `AnyType<T>` class defined in Figure 7 is an example of a smart pointer abstraction and it is this class that is specialized in defining the element templates. Hence, each element of a **SEQUENCE**, **SET**, or **CHOICE** is effectively represented as an **ANY** restricted to a single type `T`.

3.5. Implicit and Explicit Tag Modifiers

Tag modifiers have an implact on how much tag information an encoding mechanism must encode to delineate the type of a value. A type defined to be explicitly tagged has a tag formed from the concatenation of an explicitly provided tag and the tag of the immediate type being extended. A type defined to be implicitly tagged has a tag that overrides the tag of the immediate type being extended. The `IMPLICIT` and `EXPLICIT` templates facilitate the instantiation of types that may be explicitly or implicitly tagged.

The templates correspond to the **EXPLICIT** and **IMPLICIT** tag modifiers defined by ASN.1. The definition of the `EXTERNAL` type in Figure 8 illustrates the definition of both implicitly and explicitly tagged types.

Class inheritance provides a natural mechanism for expressing explicit and implicit tagging. An **IMPLICIT**ly tagged type `T` is instantiated using an `IMPLICIT` template that inherits from the base type `T` provided in the template parameter list, overriding the tag method so that the implicit tag overrides the inherited tag. Similarly, the `EXPLICIT` template is defined to inherit from the `SeqOfType<T>` template, thereby overriding the universal tag of a **SEQUENCE**, but inheriting the constructed form Our goal in implementing the set of ASN.1 types was to cleanly separate syntax from encoding/decoding semantics. Unfortunately, it is impossible to do this completely because an **EXPLICIT** modifier requires that the explicitly tagged type be encoded as a constructed type, which is effectively equivalent to encoding a type T as though it were an implicitly tagged **SEQUENCE** containing a single element of type `T`.

4. POLYMORPHIC PRESENTATION ELEMENTS

Given that arbitrary ASN.1 type specifications can be translated into a set of target language type definitions, the second objective is to permit the mapping of any ASN.1 type into a polymorphic *presentation element* (PE) that represents a particular transfer syntax encoding. We take the view that this mapping is best represented as *type conversion*, either implicit of explicit, between an ASN.1 type definition and a presentation element type representing an encoding, such as the BER. A unique feature of this approach is that it becomes relatively trivial to define and use different encoding forms; one simply defines an `PE<T>` type parameterized with the type of the encoding rules. The `encode` and `decode` methods defined in the abstract `AsnType` class are provided implementations by the various derived classes to encode and decode themselves using a `MetaPE` type, which is an abstract base class for an arbitrary `PE<T>` type (e.g., `PE<BER>`). The key idea is that any of the ASN.1/C++ types can be converted into a `PE<T>` instance that embodies a particular set of encoding rules. For example, we are investigating the definition of a `PE<ISODE>` type that would allow us to encode/decode the ASN.1 objects into/from the intermediate representation required by the ISODE presentation layer [15], which will facilitate building C++ applications using ISODE.

The C++ types generated from the ASN.1 `PasswordLookup` example presented previously in Figure 1 are encoded and decoded into presentation elements that are defined by a particular set of encoding rules. The mechanisms to encode and decode the ASN.1 object to an appropriate encoded form is handled automatically in the C++ object representing an ASN.1 type by the implementations of the `encode` and `decode` methods defined for the various classes. The choice of which encoding rules to apply are controlled by the programmer by instantiating a `PE<T>` instance parameterized by a type identifying the encoding rules to apply. Figure 9 presents a short example, using the type definitions from Figure 1, of the manner in which the ASN.1 types are encoded and decoded using a BER presentation element.

```
#include "Passwd.h"

using namespace PasswordLookup;          //  open namespace

void contrived()                                                    5
{
   Passwd pw;                            //  an ASN.1 Passwd object
   PE<BER> arg, result;                  //  two BER presentation elements

   UserName user = "postmaster";                                    10

   user.encode(arg);           //  encode argument using the BER

   if (lookupUser(arg, result) == NOTOK)
         return error("lookupUser");                                15

   pw.decode(result);          //  decode result using the BER

   printf("home dir for %s is %s\n", (char*) pw.name, (char*) pw.dir);
   ...                                                              20
};
```

Figure 9. Example use of ASN.1/C++ types.

4.1. A Polymorphic Presentation Layer

The simple example in Figure 9 has a more realistic counterpart. A presentation service access point (PSAP) defines an interface to the OSI presentation layer. A `PSAP<T>` object may be defined that takes as its type parameter an encoding type, which drives the instantiation of a presentation protocol machine that is parameterized by a set of encoding rules. For example, assuming that the P-DATA.REQUEST primitive of the presentation service is defined to take an **ANY** as its argument, different `PSAP` types can be defined that instantiate a presentation services that are parameterized by a particular set of encoding rules. Unlike some commercial implementations of the OSI presentation layer, a presentation layer built using a parameterized type mechanism can support a variety of encoding rules, such as the BER, the packed encoding rules (PER), Huitema's fast encodings [6], XDR [12] (provided that the ASN.1 types are restricted to a sensible subset), or some customized set of encoding rules (e.g., for encryption).

```
PSAP<BER> psap1;      // a PSAP to do BER encoding
PSAP<PER> psap2;      // a PSAP to do PER encoding
ANY any;

... // set up 2 associations, negotiating transfer syntax

psap1.PDataRequest (any);  // send data using BER
psap2.PDataRequest (any);  // send data using PER
```

Table 1
Encoding/Decoding Performance Comparison.

Time in seconds for 50,000 operations on a SPARC 1								
ASN.1 Type	Sun/XDR		C++/XDR		C++/BER		ISODE/BER	
	enc	dec	enc	dec	enc	dec	enc	dec
INTEGER	0.129	0.116	0.483	0.452	0.788	0.766	2.20	1.97
STRING	0.701	0.500	1.054	0.994	0.785	1.24	2.45	2.44
SEQUENCE	1.00	0.636	1.78	2.09	2.00	2.31	7.09	6.97
CHOICE	0.745	0.525	1.47	1.35	1.01	1.64	3.62	3.67

4.2. Performance Comparison

Table 1 illustrates performance data obtained from a comparison of the encoding and decoding times on a SPARC 1 of Sun XDR, an implementation of a `PE<XDR>` type that encodes/decodes our C++ ASN.1 types, our C++ BER implementation, and ISODE's BER implementation (using the *pepsy* tool). The data was collected using a set of programs that iteratively encoded and then decoded the data in a tight loop, without transmission over the network. The numbers represent the time in seconds required to encode/decode 50,000 constructs. The **SEQUENCE** and **CHOICE** consist of an **INTEGER** and an **OCTET STRING**. A complete description of the performance study and an analysis of the results is presented in [13]. The data demonstrates that the C++ implementation has very competitive performance and that the flexibility obtained by separating the ASN.1 representation from the encoding/decoding rules embodied in a `PE<T>` object does not adversely affect the performance. We are encouraged by these results. We have already identified an inefficiency related to tag checking that is being corrected, which we expect will improve the decoding performance for all types. Our plan is to perform a more comprehensive performance analysis of different `PE<T>` encoding forms using Huitema's proposed ASN.1 benchmark [5].

5. SUMMARY

The translation of ASN.1 into C++ is but one aspect of an object-oriented protocol framework using polymorphic type structures and inheritance. This framework is the result of an experiment at *objectifying* the structure of the upper layers of the ISO Reference Model [10], specifically focused on providing a flexible communication infrastructure for concurrent object-oriented applications in a multi-protocol environment. The goal of this framework is to create an extensible and efficient set of abstractions that facilitate the instantiatiation of multiple and diverse sets of communication services within an application [17]. This object-oriented protocol development framework is called OOSI (*ooo-zi*). Similar research-oriented upper layer protocol development environments that we are familiar with include ELROS [2], OTSO [11], DAS [14], and ISODE [15]. OOSI was influenced by ISODE, but is engineered for maximum flexibility and performance. OOSI is similar to OTSO in that both utilize object-oriented features, such as inheritance, to structure protocol layers. OOSI, however, exploits type polymorphism to gain compositional flexibility.

The C++ ASN.1 type abstractions presented here are the most critical part of the presentation layer framework and these abstractions advance the state of the art for implementing application layer services. The expressive power of object-oriented type systems suggest that new thinking be applied to the implementation of the presentation layer for those applications that are amenable to an object-oriented formulation. A useful conclusion to be drawn from this work is that the data representation aspect of presentation layer services are directly expressible in a language type system that supports the definition of polymorphic types. This conclusion suggests that perhaps the traditional concept of a presentation layer can be subsumed by a rich language type system.

The concepts underlying the ASN.1/C++ abstractions are similar to ideas in ELROS of embedding ASN.1 syntax in the C language; however, our work is distinct from the preprocessing strategy of ELROS in that inherent language type definition mechanisms are used to effectively embed ASN.1 into native C++. Advantages of our approach to programming with ASN.1 over others include conceptual uniformity, extensibility, and simplified debugging due to a nearly one-to-one correspondence between an ASN.1 specification and user-defined types.

A translator from ASN.1 to the C++ class templates, called *caty*, is currently working and a wide range of standard ASN.1 specifications have been translated. The class template implementation significantly reduces the amount of work required by an ASN.1 translator since the C++ compiler assumes much of the responsibility for generating the proper types and dispatching of virtual `encode`, `decode`, and `tag` methods.

This work is not limited to applications based on the upper layer OSI protocols. The ASN.1 types are independent from the protocol layers. It is our hope that the work described here will prove useful in the development of real applications. In particular, this work has great applicability in those applications that define presentation services directly over a transport service; for example, the Internet Simple Network Management Protocol (SNMP) [3], the ANSI Z39.50 information retrieval protocol [1] used in implementations of the Wide Area Information Service (WAIS) [4], and lightweight client protocols for accessing directories or message stores.

ACKNOWLEDGEMENTS

We appreciate the assistance of Rajesh Khera, who participated in an early implementation of the ASN.1 class hierarchy. Wendy Long implemented CATY using the ASN.1 grammar from the ISODE *pepsy* tool [15]. The book by Steedman [16] was helpful in clarifying certain aspects of ASN.1 and the BER encodings.

REFERENCES

1. American National Standards Institute. *Z39.50 Information Retrieval Service Definitions and Protocol Specification for Library Applications*, July 1992.
2. M. L. Branstetter, J. A. Guse, and D. M. Nessett. ELROS — an embedded language for remote operations service. In *1992 IFIP International Working Conference on Upper Layer Protocols, Architectures and Applications (ULPAA'92)*, pages 33–47. Elsevier/North-Holland, 1992.

3. J. Case, M. Fedor, M. Schoffstall, and C. Davin. Simple Network Management Protocol (SNMP). Internet Protocol Specification RFC 1157, Network Information Center, SRI International, May 1990.
4. Franklin Davis, Brewster Kahle, Harry Morris, Jim Salem, and Tracy Shen. WAIS interface protocol prototype functional specification. Technical Report version 1.5, Thinking Machines Corporation, April 1990.
5. Christian Huitema and Ghislain Chave. Measuring the performance of an ASN.1 compiler. In *1992 IFIP International Working Conference on Upper Layer Protocols, Architectures and Applications (ULPAA'92)*, pages 99–112. Elsevier/North-Holland, 1992.
6. Christian Huitema and Assem Doghri. Defining faster transfer syntaxes for the OSI presentation protocol. *ACM Computer Communication Review*, 19(5):44–55, October 1989.
7. Christian Huitema and Assem Doghri. A high speed approach for the OSI presentation protocol. In H. Rudin and R. Williamson, editors, *Protocols for High-Speed Networks*, pages 277–287. Elsevier/North-Holland, 1989.
8. International Organization for Standardization. *Information Processing — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1)*, 1987. ISO 8824 (CCITT X.208).
9. International Organization for Standardization. *Information Processing — Open Systems Interconnection — Specification of Basic Encoding Rules for Abstract Syntax Notation One*, 1987. ISO 8825 (CCITT X.209).
10. International Organization for Standardization. *Information Processing — Open Systems Interconnection — Basic Reference Model*, 1987. ISO 7498.
11. Juha Koivisto and Juhani Malka. OTSO—an object-oriented approach to distributed computation. In *USENIX C++ Conference Proceedings*, pages 163–177, April 1991.
12. Sun Microsystems. XDR: External data representation standard. Internet Protocol Specification RFC 1014, Network Information Center, SRI International, June 1987.
13. Robert W. Mullins. Separating representations and translation of shared data. M.S. project report, Virginia Polytechnic Institute and State University, Department of Computer Science, September 1993.
14. Gerald Neufeld. *Distributed Application Support Package User Manual.* University of British Columbia. Draft.
15. Marshall T. Rose. *The ISO Development Environment User's Manual, Volumes 1-5.* Performance Systems International, July 1991. Version 7.0.
16. Douglas Steedman. *Abstract Syntax Notation One (ASN.1): The Tutorial and Reference.* Technology Appraisals, 1990.
17. Christian Tschudin. Flexible protocol stacks. In *SIGCOMM'91 Conference Proceedings*, pages 197–205, September 1991.