CATY:
An ASN.1-C++ Translator
in support of
Distributed Object-Oriented Applications

by
Wendy Long

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      Marc Abrams

April, 1994
Blacksburg, Virginia
CATY:
An ASN.1-C++ Translator
in support of
Distributed Object-Oriented Applications

by
Wendy Long
Dennis G. Kafura, Chairman
Computer Science

Abstract

When heterogeneous computers exchange data over a network, they must agree on a common interpretation of the data. The OSI suite of protocols includes a standard notation, Abstract Syntax Notation One (ASN.1), for describing the structure ("abstract syntax") of data. Previous work has shown that C++ is a good language for work with layered network architectures and specifically with ASN.1: the inheritance and polymorphism features of C++ are nicely suited for work with layered protocols, which can be seen and used in object-oriented terms; a C++ class hierarchy, designed to capture the language concepts of ASN.1, successfully separates the abstract syntax (or application level) from the encoding used during transfer (the "transfer syntax" at presentation level); and the class construct and scoping rules of C++ and the design of the class hierarchy much better preserve the structure and content of ASN.1 than do past attempts with C. This report presents CATY (Class-oriented ASN.1 Translator, Yacc-based), a translator from ASN.1 to a corresponding C++
abstract syntax class hierarchy. It is shown in this report that the translations produced by CATY are preferable to those produced by other translators based on the following criteria: preservation of names and types, consistent access to elements, support of modularity and subtypes, resolution of forward references, flexibility of encoding, and generality of use. Furthermore, it is shown that CATY has better throughput than PEPSY, an ASN.1 to C translator from ISODE.
Table of Contents

1 Introduction ................................................................................................................. 1
1.1 Why Use C++ as a Target Language? ................................................................. 1
1.1.1 In Defense of Object-Oriented Network Design ........................................... 3
1.1.2 Separation of Abstract from Transfer Syntax .............................................. 4
1.1.3 Compatibility between ASN.1 and C++ ......................................................... 4
1.2 A Brief Introduction to ASN.1 ............................................................................ 5
1.2.1 Modules ........................................................................................................ 5
1.2.2 Types ........................................................................................................... 7
1.2.3 Tags .............................................................................................................. 11
1.3 Previous Work ..................................................................................................... 12
1.3.1 ASN.1 to C Translators .............................................................................. 13
1.3.2 ASN.1 to C++ Translators .......................................................................... 17
1.4 Justification, and Introduction to the Rest of the Report ................................. 20
2 The Translation ....................................................................................................... 22
2.1 Modules ............................................................................................................ 22
2.2 Type Definitions ............................................................................................... 23
2.2.1 Primitives, Usefulness, and User-Defined Types ....................................... 23
2.2.2 Constructed Types ....................................................................................... 25
2.2.2.1 REQUIRED, OPTIONAL, and DEFAULTS ........................................ 26
2.2.2.2 Nested Types and Missing Names ......................................................... 28
2.2.3 Meta-Structural Types ............................................................................... 29
2.2.4 Subtypes ...................................................................................................... 30
2.3 Tags ................................................................................................................ 30
List of Figures

Figure 1 - Abstract syntax and transfer syntax on a distributed system with heterogeneous machines ................................................................. 2
Figure 2 - The OSI 7-layer model ........................................................................ 3
Figure 3 - ASN.1 data types [6, 10, 18] .................................................................. 8
Figure 4 - Summary of ASN.1 translator attributes .............................................. 21
Figure 5 - The ASN.1 grammar .............................................................................. 33
Figure 6 - Definition of Type_Element .................................................................. 34
Figure 7 - CATY vs. PEPSY as tested on three ASN.1 modules, based on processor clock ticks used by each program ......................................................... 48
Figure 8 - Translation, linking, and execution of the PasswordLookup ASN.1 specification ........................................................................................................... 50
Figure 9 - Types and type attributes tested with each ASN.1 source file .......... 51
1 Introduction

When heterogeneous computers exchange data over a network, they must agree on a means for interpreting the structure of the data, and for representing the data in transit. The OSI suite of protocols solves this problem by including a standard notation, Abstract Syntax Notation One (ASN.1), for describing the structure of data, and a standard transfer syntax, the Basic Encoding Rules (BER), for representing the data. The two are often used together, as follows (refer to Figure 1). A distributed application, such as an airline reservation system, has a standard data structure, such as a record containing the name and address of the client, the airline, flight number, seat reserved, and so on. The data structure relevant to the application is agreed upon by all users of the application, regardless of their local machines or programming languages. This data structure would be defined, in the form of type definitions, in ASN.1. This type definition would then be translated, in each machine, into the local language, for use by the application program. A library of encoding routines and decoding routines would also be made available, so that instances of data of this type can be transformed between the physical representation used by the local machine, and the agreed upon transfer syntax, for instance BER. As this process implies, the uses of abstract and transfer syntaxes are separate issues, and should be dealt with separately.

1.1 Why Use C++ as a Target Language?

This report presents CATY (Class-oriented ASN.1 Translator, Yacc-based), a translator from ASN.1 to a corresponding C++ abstract syntax class hierarchy. The advantages of using C++ as the target language for an ASN.1
translator are as follows:

1. The inheritance and polymorphic features of C++ are nicely suited for work with layered network protocols, which can be seen and used in object-oriented terms [9, 11].

2. The C++ class hierarchy successfully separates the abstract syntax from the transfer syntax, thus keeping the implementation of transfer syntax transparent to the user as well as allowing alternative transfer syntaxes to be negotiated at runtime [13].

3. The nature of C++ and the design of the class hierarchy much better preserve the structure and content of ASN.1 than do past attempts with C [10].

These three factors make C++ a good language for work with layered network architectures and specifically with ASN.1. Each factor is discussed in depth in the sections following.
1.1.1 In Defense of Object-Oriented Network Design

Layered network architectures, such as the OSI protocol suite, lend themselves nicely to an object-oriented paradigm [11]. Figure 2 shows the OSI seven-layer model. Each layer is designed to provide services to the layer above it, and can use the layers below only in terms of the well-defined interface of the layer directly below. Peer layers at different sites communicate using objects called protocol data units (PDU's). Implementation of services provided by each layer is hidden from the layers above and below. Koivisto and Reilly [9] draw a parallel between this model and the object oriented nature of C++. An entity at one layer, for instance, a presentation element, can be defined as a class, and elements of other layers can use this element by inheriting from it. Thus, implementation of the presentation element can be hidden in the private section of the class in which it is defined, while the interface can be defined in the public section and used by other layers when inherited. Koivisto and Reilly further propose that an entity's relationships with its peers and with its upper and lower neighbors can be modeled using multiple inheritance: the entity's class
inherits multiply from the layer above, from the layer below, and from an appropriate PDU class.

1.1.2 Separation of Abstract from Transfer Syntax

Distributed data type definitions and the encoding of those types have often been treated as the same issue, with "ASN.1" being used to refer to both the abstract syntax notation and the Basic Encoding Rules. Translators are often written that will not only translate between ASN.1 and the local language, but also inseperably include routined to encode and decode the data using BER. This is not always practical, since BER is not necessarily the desirable encoding format (Xerox's Courier protocol and Sun Microsystems' XDR are possible alternatives). Furthermore, the abstract syntax and the encoding and decoding of data are distinct tasks that should ideally be separated from each other. A pair of C++ class hierarchies, one for the abstract representation of data and one for the encoding and decoding, has been developed by Mullins, Lavender and Kafura [10, 13]. This work not only separates the issues of the abstract syntax from the encoding and decoding, but also allows applications to be written using the object-oriented paradigm. CATY, the translator presented in this report, completes a marriage of layered protocols with this C++ class hierarchy by translating ASN.1 into the class hierarchy.

1.1.3 Compatibility between ASN.1 and C++

Several ASN.1 to C translators have been written [3, 7, 14, 16]. They will be discussed in greater detail in section 1.3. As will be seen, these translators not only confuse the distinction between abstract and transfer syntax, but also
produce C translations that employ bad programming practices, such as a loss of much of the type information and a confusing mix of compiler directives and pointers in the midst of the type definitions. Use of C++ as the target language can solve these problems by way of strong type checking and the use of templates [10, 11]. The specific mappings from ASN.1 to C++ will be discussed in great detail in chapter 2.

1.2 A Brief Introduction to ASN.1

This section provides a brief introduction to ASN.1. For a more thorough description, the reader may refer to the tutorial written by Steedman [18].

As an abstract syntax, ASN.1 is used to describe the structure and content of application information. Thus, definitions of types are the fundamental building blocks of ASN.1. The following discussion will describe the organization of type definitions into modules, the varieties and structure of type definitions, and ways to modify and add to the predefined ASN.1 types.

1.2.1 Modules

A module is a named collection of type (and occasionally value) definitions. A module begins with an identifier, the keyword DEFINITIONS, an optional default style for tags (to be discussed later), the assignment symbol "::=", and the module body. The module body is made up of the keyword BEGIN, EXPORTS and IMPORTS statements (if any), the type and value assignments, and the keyword END. The following is an example of an ASN.1 module:
Sample-Module DEFINITIONS ::= 
BEGIN 

EXPORTS  Type1, Type2; 
IMPORTS  OtherType FROM Other-Module; 

Type1 ::= SEQUENCE { 
        Type2, 
        Type3 
    } 

Type2 ::= INTEGER 
Type3 ::= SET OF OtherType 

... 
   -- More type definitions 
...

END 

Notice that names in ASN.1 may contain any alphanumeric characters and the hyphen, and the first character in a name must be alphabetical. Notice also that comments are indicated by "--" at the beginning of a line.

The EXPORTS statement indicates to which types, defined in the current module, other modules may refer. Types not listed in the EXPORTS statement are thus encapsulated within the module. However, if there is no EXPORTS statement, all types defined in the module are visible to other modules. Similarly, the IMPORTS statement indicates to which types, from other modules, this module will refer. It is also possible to refer to types from other modules without specifying this in an IMPORTS statement: one can simply write a cross-module reference, <ModuleName>.<TypeName>, to refer to type <TypeName> defined in module <ModuleName>.

The rest of the module contains type definitions. Type definitions may occur in any order in the module, since there is no rule concerning forward references in ASN.1. It is conventional, however, to order type definitions in a top-down fashion. Type definitions may also refer to themselves recursively.
1.2.2 Types

Figure 3 lists the types available in ASN.1. There are three kinds of built-in types: primitive (or simple), constructed (or structured), and meta-structural types. Additionally, there are currently four "useful types" that are not technically built-in but are treated as if they are. Type definitions take the form `<TypeName>::= <Type>`, where `<Type>` is any built-in or useful type, any name of a type defined in the current module or imported, or any cross-module reference to a type defined elsewhere.

The primitive types are described at the top of Figure 3. They are: INTEGER, REAL, NULL, BOOLEAN, ENUMERATED, BIT STRING, OCTET STRING, OBJECT IDENTIFIER, and a collection of character string types. The character string types are: NumericString, PrintableString, TeletexString (T61String), VideotexString, VisibleString (ISO646String), IA5String, GraphicString, and GeneralString. INTEGER and BIT STRING types may be further specified with enumerated names for specific values or bits. For example, the following examples of primitive types are from the ASN.1 specification of FTAM [8]:

```plaintext
Concurrence-Key ::= BIT STRING {
  not-required (0),
  shared (1),
  exclusive (2),
  no-access (3)
}

State-Result ::= INTEGER {
  success (0),
  failure (1)
}
```
<table>
<thead>
<tr>
<th><strong>Primitive Types</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
</tr>
<tr>
<td>REAL</td>
</tr>
<tr>
<td>NULL</td>
</tr>
<tr>
<td>BOOLEAN</td>
</tr>
<tr>
<td>ENUMERATED</td>
</tr>
<tr>
<td>BIT STRING</td>
</tr>
<tr>
<td>OCTET STRING</td>
</tr>
<tr>
<td>Character Strings</td>
</tr>
<tr>
<td>OBJECT IDENTIFIER</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Constructed Types</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQUENCE</td>
</tr>
<tr>
<td>SEQUENCE OF</td>
</tr>
<tr>
<td>SET</td>
</tr>
<tr>
<td>SET OF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Meta-Structural Types</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>CHOICE</td>
</tr>
<tr>
<td>ANY</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Useful Types</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>GeneralizedTime and UniversalTime</td>
</tr>
<tr>
<td>EXTERNAL</td>
</tr>
<tr>
<td>ObjectDescriptor</td>
</tr>
</tbody>
</table>

Figure 3: ASN.1 data types [6, 10, 18]
This notation allows an identifier to be associated with a specific integer value or bit position. For example, an instance of type State-Result may now be assigned the values “success” or “failure” as if they were integer values; and instances of type Concurrency-Key may use the enumerated names to refer to specific bit positions, such as “not-required” for bit zero. These enumerations do not restrict the range of possible values that instances of the type can take (i.e., values other than the named ones are still allowed).

The constructed types, SEQUENCE, SEQUENCE OF, SET, and SET OF, are described in the second section of Figure 3. Constructed types are defined in terms of other types, their component types. The component types may be any type: primitive, user-defined, constructed, etc.; nesting of constructed types is unlimited. Component types are conventionally given names, but names for component types may be omitted. Component types may also be OPTIONAL, or may have DEFAULT values. The following SEQUENCE is from FTAM [8]:

\[
\text{F-ERASE-response ::= SEQUENCE }
\]
\[
\text{action-result Action-Result DEFAULT success,}
\]
\[
\text{diagnostic Diagnostic OPTIONAL}
\]

In this example, the component named "action-result" of type "Action-Result" takes a default value of "success" when no value has been assigned, whereas the component named "diagnostic" need not be assigned a value at all in a valid occurrence of the SEQUENCE. Notice that, by convention, component names and value names are not capitalized, whereas type names are.

Meta-structural types, as described by Lavender, Kafura, and Mullins [10], include CHOICE and ANY, and can take one of a choice of several types (the choice being from all the possible types in the case of ANY). For each CHOICE
element it is necessary that each alternative have a different tag (tags are explained below) so that the alternative choices can be easily distinguished, for example:

```
DistinctChoices ::= CHOICE {
    first-choice   [1] INTEGER,
    second-choice  [2] INTEGER,
    third-choice   [3] REAL
}
```

In this example, instances of the type DistinctChoices can take on values of INTEGER or REAL types, with "first-choice" and "second-choice" values distinguished by the tags. The tags are shown in brackets; for example, "second-choice" has tag "[2]".

In addition to the primitive, structural, and meta-structural types are the "Useful" types shown at the bottom of Figure 3. They are not technically primitive types, although they are treated as such. The Useful types currently include GeneralizedTime and UniversalTime, which represent date and time values; EXTERNAL, similar to extern in C, which imbeds a data type defined elsewhere; and ObjectDescriptor, which holds information that can be found in the header of a module.

Several methods of subtyping are available in ASN.1. Subtypes allow the designer to restrict a type's range, for instance,

```
Celsius    ::= INTEGER (1..100)
Weekend    ::= Days-of-Week (saturday | sunday)
```

In this example, an instance of type "Celsius" can only take on an integer value from 1 to 100, inclusive, and an instance of type "Weekend" has only two possible values: "saturday" or "sunday".

1.2.3 Tags

Tags are a means for adding to or modifying the information conveyed by an ASN.1 type. Tags are relevant primarily to the encoding and decoding of instances of the type, and so do not ideally belong in the realm of abstract syntax. However, tags are nevertheless defined as part of ASN.1 and thus must be addressed. We have already seen a use for tags: to distinguish among CHOICE elements in a CHOICE type definition. Including the tags in "DistinctChoices" (see Section 1.2.2) has the effect of encoding the first and second choices differently from the usual INTEGER encoding, so that when an INTEGER instance of this CHOICE type is decoded it can be determined whether the value is of type "first-choice" or "second-choice". Tags are also useful for resolving ambiguity when there are two or more OPTIONAL component types of the same base type within a SET or SEQUENCE.

A tag consists of a class and a (non-negative integer) number. There are four classes: UNIVERSAL, APPLICATION, CONTEXT, and PRIVATE. All ASN.1 built-in types have implied UNIVERSAL tags, with numbers specified by the ASN.1 standard. Tags of the other three classes may be used by designers to distinguish new user-defined types from the built-in types with UNIVERSAL tags. APPLICATION tags are used to distinguish types within an entire application; CONTEXT tags (as seen in "DistinctChoices") distinguish types within a single type definition; and PRIVATE tags are used to facilitate private use and
modification of an abstract syntax controlled by an organization, such that the private application will not conflict with the general one. Numbers used with these three classes of tags are arbitrary except as agreed upon by the users of the ASN.1 specification. A tag can occur anywhere a type can, either in the outer level of a type definition, or as a modifier of a component of a constructed type. Some examples of tag usage are:

\[
\begin{align*}
\text{OurOwnInt} & := \text{[PRIVATE 5] INTEGER} \\
\text{Airlint} & := \text{[APPLICATION 8] INTEGER} \\
\text{RegularInt} & := \text{INTEGER}
\end{align*}
\]

All three of the above integers are distinct, because they will be encoded with different tags ("RegularInt" has an implied [UNIVERSAL 2]) and will thus be distinguishable when decoded.

Tags are either IMPLIED or EXPLICIT. If neither keyword is specified for a tag (between the ending "]") and the base type) then the tag takes on the tag style default specified at the beginning of the module (either "IMPLIED TAGS" or "EXPLICIT TAGS"). If no default is given for the module's tag styles, then EXPLICIT is assumed. The difference between EXPLICIT and IMPLIED is that IMPLIED tags are encoded instead of the UNIVERSAL tag of the base type, whereas EXPLICIT tags are encoded in addition to the base type's tag.

1.3 Previous Work

This section describes previous ASN.1 translators, including both ASN.1 to C and ASN.1 to C++ translators.

Work presented here will be judged based on the following criteria:

- **name preservation**: are user-defined names preserved, or changed
(“mangled”) by the translator?

- **type preservation**: are type attributes preserved, or are some lost in the translation?

- **consistent access**: is the access to an element kept generally consistent, or is indirection introduced (via pointers)?

- **modularity**: is the encapsulation of types preserved, or are all definitions EXPORTED?

- **forward reference resolution**: are the (customarily) forward references in ASN.1 modules translated into the correct order for the target language?

- **subtypes**: are the subtyping constructs of ASN.1 implemented in the target language?

- **flexibility of encoding**: are encoding rules treated separately from the abstract syntax, and are alternate encoding rules available?

- **environment limitation**: is the generated code limited to a specific run-time environment, or can it be used in a variety of environments?

It will be shown that CATY addresses the above issues better than any of the previous translators.

### 1.3.1 ASN.1 to C Translators

**CASN1**

CASN1 is an ASN.1-C translator developed by Neufeld and Yang [14]. It translates an ASN.1 module definition into corresponding C type definitions, and for each type defined in the ASN.1 module, BER encoding and decoding routines are also generated. User-defined names are generally preserved, although when a name is not specified for a type (this unfortunate tendency to leave out identifiers for members of constructed types can be found in the ASN.1 definitions of FTAM [8] and SNMP [5]) the generated name, 'T' followed by digits, is not descriptive. Much of the type information is lost: primitive ASN.1 types
INTEGER, REAL, ENUMERATED, and NULL are translated into C types int, float, enum, and int, respectively; ASN.1 constructed types SEQUENCE {...}, SET {...}, and CHOICE {...} are all translated into the C type struct {...}. The original types are indicated only in comments inserted by the translator. Thus, the resulting types do not correspond by name or even uniquely to the original ASN.1 type; this causes some type attributes to be lost in the translation. More type attributes are lost when tags in the C code are dealt with simply by #define's, not associated directly with the types they modify except for a similarity in the naming. Access is not consistent: the OPTIONAL construct in ASN.1 is implemented in C by generating a pointer to the optional type, instead of the type itself. If the pointer is null, this indicates that the OPTIONAL element is not defined. Thus, the only indication of whether an item is considered OPTIONAL is a "*" before the item’s name (and translator-generated comments). This can be easily confused with other types that are always pointers, such as BITS and OCTS, the equivalent of ASN.1 BITSTRING and OCTETSTRING. In addition, the varying levels of pointers resulting from an OPTIONAL BITSTRING add to the complexity of data access. CASN1 does not support modularity. It also relies on the assumption that the ASN.1 types in each module will be defined top-down, as is customary in ASN.1. This could cause forward reference problems if type definitions are defined in any other order. Encoding rules are not flexible at all, since BER routines are automatically generated for each type. Two advantages of CASN1 are that it is able to preserve self-referential structures, which are allowable in C; and that it is intended for general use.
MAVROS

MAVROS, developed at INRIA [3], is a C programming environment that facilitates use of ASN.1 specifications. The environment contains an ASN.1-C translator (MAVCOD+MAVROS), an ASN.1 library for runtime support, and a mechanism for user-defined extensions of the usual translation from ASN.1 (MAVROS). The straightforward translator provided in the MAVROS environment has many of the same weaknesses as CASN1: type attributes are lost in the same ways, through the use of struct and #define's; unexpected indirection of access is introduced when OPTIONAL elements are defined as pointers; and the encoding rules are fixed at compile-time. Furthermore, although modularity is partially supported, it is supported by way of a consistent name-mangling that causes the original names never to be preserved: all type names are prefixed with the module name. For types that do not have names in the original ASN.1, MAVROS generates an "x_" followed by an integer describing the type's position in its parent type. This naming device is not descriptive.

MAVROS's advantages are its extensibility, its (limited) support of modularity, and the fact that it is intended for general use. However, while the user-defined extensibility of the translation is a useful tool, it is awkward to use: the user must essentially redefine all the original type definitions by hand, and then run the new definitions through the MAVROS processor for merging with the ASN.1 specification and for production of the encoding and decoding routines. These extensions could be more easily made if the target language were C++, where inheritance is an automatic feature and no translation is necessary. The extensibility feature of MAVROS also ties the BER encoding and decoding routines inextricably to the abstract type definitions.
PEPSY

PEPSY is an ASN.1-C translator developed by Rose [16] for the ISO Development Environment (ISODE). PEPSY was made for use specifically with ISODE, and takes as input ASN.1 modules that have been preprocessed by other programs in ISODE (the input is close to standard ASN.1). In addition to mangling user-defined names with several prefixes and suffixes each, PEPSY loses type attributes in ways similar to those of CASN1 and MAVROS. Constructed types are all translated into struct's, and more than one primitive type can map onto the same type in C. Tags are again treated as #define's. OPTIONAL items are handled in several different ways, depending on the type; although this is an attempt to preserve the access method by inventing different ways to deal with non-pointer OPTIONAL types, it only serves to make OPTIONAL's more confusing.

PEPSY is, however, an improvement over CASN1. Like MAVROS, PEPSY provides some support of modularity by prefixing all names with the module name. PEPSY does not assume an order to the ASN.1 type definitions, but rather reorders the definitions as necessary to avoid unresolved forward references. Finally, PEPSY allows the user to specify names of encoding and/or decoding routines within the ASN.1 definition; unfortunately these routines must follow very rigid and elaborate calling and return value conventions [16] and are fixed once the modified ASN.1 module has been translated.
1.3.2 ASN.1 to C++ Translators

Since C++ is a relatively newer language, there are fewer ASN.1-C++ translators. Two such efforts are discussed in this section, and then the translator developed in this project is introduced.

CLASN

Koivisto and Reilly [9] developed CLASN, which, when used along with OTSO's compiler "prepro" [20], generates C++ code. For each module, primitive ASN.1 types and most of the constructed types are translated to corresponding C type definitions, which are placed in a header file for type definitions; corresponding encoding and decoding routines are placed in a code file. (The specifics of this process are not explained by Koivisto and Reilly [9]). Each ASN.1 module is, however, translated into its own C++ class (with the class name the same as the ASN.1 module name). Certain constructed type definitions are marked by "$ENTRY" beforehand, supposedly by the user, indicating that these are PDU types. Each type definition thus marked becomes a virtual member function of the class, with the user-defined name preserved, and with elements of the constructed type becoming parameters to the function. This implies that there will be some function (defined by the user) associated with each "$ENTRY" type, to be bound dynamically to the translated virtual function, and that the preservation of type attributes depends on the user's definition of the function. Once CLASN has generated these virtual functions, OTSO's "prepro" compiler combines these virtual functions with user-written routines (defining the types and adding encoding and decoding rules) to produce the complete C++ code. Modularity is supported in that each ASN.1 module
becomes a class, but not all types originally defined in the ASN.1 module are members of that class. Issues of forward references and subtyping are not addressed by Koivisto and Reilly, and CLASN can be used only with OTSO.

**SNACC**

SNACC, developed by Michael Sample [17], generates ASN.1 module translations into either C or C++, along with BER encoding and decoding routines. SNACC is intended for general use and, with some exceptions, is quite similar to CATY. User-defined names are preserved except that hyphens are changed to underscores, and names conflicting with C or C++ keywords have integers appended. While the C type translations have problems similar to those of the other ASN.1-C translators discussed above, the C++ translations attempt to make use of the object-oriented features of C++. Each translated type inherits from the type AsnType, which is defined more specifically for each primitive type and to which members are added for each constructed type. The small class library from which the types inherit includes only AsnType and similarly named types (AsnInt, AsnBool, etc.) corresponding to the primitive ASN.1 types. Unfortunately all constructed types inherit directly from AsnType. Thus, original ASN.1 type attributes are lost either partly or altogether in the translation. SNACC introduces indirection when there is an OPTIONAL element of a constructed type by making the element a pointer (without adding comments to the effect that this pointer is for the purpose of an OPTIONAL element). SNACC supports modularity, although the modules that refer to each other must be compiled together by SNACC in order of their dependence. Type definitions are painstakingly reordered to eliminate forward references, and are checked for
illegal recursive definitions. Unlike any of the translators mentioned previously, SNACC supports the ASN.1 subtype constructs. Unfortunately, SNACC attaches BER encoding and decoding routines to each type at compile time, and embeds tag values within these routines.

CATY

CATY, the translator developed in this project is intended for general use in C++ programming, together with the C++ abstract syntax class hierarchy developed by Mullins, Lavender, and Katura [10, 13]. CATY is an improvement over the previous translators from ASN.1 to C and C++ in that the C++ class hierarchy preserves the names and form of the original ASN.1. Primitive types are defined as typedef's corresponding to types defined in the class hierarchy by the same name as the ASN.1 type. Constructed types are translated to C++ classes, which simply inherit from classes in the class hierarchy corresponding to types of the same name as the ASN.1 constructed type. Type attributes are further preserved using templates: tags are included in a template within the types they modify, rather than in separately defined and named #define's; and distinctions between REQUIRED and OPTIONAL, and between IMPLICIT and EXPLICIT, are made using templates of the same names. Since OPTIONAL types are handled by templates, the access method can be preserved for OPTIONAL types, whereas these types were turned into pointers by other translators. Module references are handled by translating each module into a class with the defined types as data members; interdependent modules need not be translated together. Type definitions are automatically reordered by CATY as needed to avoid forward references. Although subtypes are not currently
translated, work is in progress to incorporate subtypes in the C++ class hierarchy, after which subtype translation will be added to CATY. Finally, encoding rules are left as an entirely separate issue to be negotiated at runtime.

Figure 4 summarizes the above discussion of the six translators. Where the authors of the translator did not discuss an issue, "not addressed" is written in the chart. It can easily be seen that CATY encompasses all the best attributes of these ASN.1 translators.

1.4 Justification, and Introduction to the Rest of the Report

In the above discussion it has been shown that a translator from ASN.1 to C++ is a desirable and useful contribution. C++ is a more powerful language than C for use with layered network architectures, which can be represented effectively in an object-oriented paradigm; C++ is a logical match for ASN.1, because of the flexibility of style allowed by templates; and the ASN.1 class hierarchy makes it possible to remove concerns of encoding and decoding from the treatment of the abstract syntax.

The CATY translator will be described in detail in the remainder of the report. Chapter 2 describes the mapping of ASN.1 constructs into C++. Chapter 3 describes the internal structure of the translator. Chapter 4 reports on the translator's performance, and Chapter 5 draws conclusions. A "man" page is given in Appendix A.
<table>
<thead>
<tr>
<th>Attribute:</th>
<th>CASN1</th>
<th>MAVROS</th>
<th>PEPSY</th>
<th>CLASN</th>
<th>SNACC</th>
<th>CATY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target language</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C++</td>
<td>C or C++</td>
<td>C++</td>
</tr>
<tr>
<td>Name preservation</td>
<td>generally preserved</td>
<td>all names are changed</td>
<td>extreme mangling</td>
<td>generally preserved</td>
<td>generally preserved</td>
<td>generally preserved</td>
</tr>
<tr>
<td>Type preservation</td>
<td>poor</td>
<td>poor</td>
<td>poor</td>
<td>virtual functions defined by user</td>
<td>fair</td>
<td>generally preserved</td>
</tr>
<tr>
<td>Access method preservation</td>
<td>indirection introduced</td>
<td>indirection introduced</td>
<td>preserved</td>
<td>lost; specified by function definition</td>
<td>indirection introduced</td>
<td>preserved</td>
</tr>
<tr>
<td>Modularity</td>
<td>none</td>
<td>simulated via name prefix</td>
<td>simulated via name prefix</td>
<td>yes: each module becomes a class</td>
<td>supported if compiled together</td>
<td>yes: each module becomes a class</td>
</tr>
<tr>
<td>Forward reference resolution</td>
<td>based on invalid assumption</td>
<td>no</td>
<td>yes</td>
<td>not addressed</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Subtypes</td>
<td>not addressed</td>
<td>not addressed</td>
<td>no</td>
<td>not addressed</td>
<td>yes</td>
<td>yes (in progress)</td>
</tr>
<tr>
<td>Flexibility of encoding</td>
<td>no</td>
<td>no</td>
<td>user-specified at compile-time</td>
<td>user-specified at compile-time</td>
<td>no</td>
<td>user-specified at run-time</td>
</tr>
<tr>
<td>Environment Limitation</td>
<td>none</td>
<td>none</td>
<td>'ISODE'</td>
<td>OTSO</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

Figure 4: Summary of ASN.1 translator attributes
2 The Translation

This section describes the details of the mapping from ASN.1 source to C++ code. For a full discussion of the internal details of the C++ class hierarchy, the reader may refer to Mullins [13]. This section discusses only the use of the class hierarchy for translation from ASN.1.

2.1 Modules

Each ASN.1 input module is translated into an output file containing a single C++ class. Nested within this class are the translations of the module's type definitions. User-defined names are retained intact, except that hyphens are changed to underscores, and names that happen to be C or C++ keywords are appended with "_ASN". For example, the ASN.1 Sample-Module described in section 1.2 would translate as follows:

```cpp
class Sample_Module {
    public:
        /* translations of exportable type definitions */
    private:
        /* translations of non-exportable type definitions */
} /* end module Sample_Module */
```

This translation method preserves the modularity intended by the original ASN.1. Types indicated to be EXPORTS (or, by default, all the types) are accessible from other modules simply by writing "<ModuleName>::<TypeName>", since
these types are defined in the "public" region of the class. Any types intended not to be exported are, on the other hand, hidden in the "private" region.

Since types must be defined before they are referenced in C++, whereas there is no such rule in ASN.1, type definitions are reordered so that forward references are resolved, wherever possible. There are type definitions whose forward references cannot be resolved. This occurs when definitions refer to type names that are not defined within the current module or that are not indicated to be cross-module references. When forward references cannot be resolved, the types are declared and then defined at the bottom of the class beneath the comment "/* possibly undefined types */". An error message is also given, telling the user what name was not defined within each unresolved type definition.

As of this date, recursively defined types, such as binary trees, are also treated as type definitions with unresolved forward references; it is hoped that slight redefinition of some templates in the C++ class hierarchy will allow such recursively defined types to be properly handled.

2.2 Type Definitions

2.2.1 Primitives, Usefults, and User-Defined Types

Simple or primitive types, as well as the so-called "useful" types, are translated as typedefs using types from the class hierarchy of the same name as the ASN.1 types. The mapping is as follows:

<table>
<thead>
<tr>
<th>ASN.1</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyInt ::= INTEGER</td>
<td>typedef INTEGER MyInt;</td>
</tr>
<tr>
<td>MyReal ::= REAL</td>
<td>typedef REAL MyReal;</td>
</tr>
<tr>
<td>MyNull ::= NULL</td>
<td>typedef ASN_NULL MyNull;</td>
</tr>
<tr>
<td>MyBool ::= BOOLEAN</td>
<td>typedef BOOLEAN MyBool;</td>
</tr>
</tbody>
</table>
MyBitstring ::= BIT STRING
typedef BIT_STRING MyBitstring;
MyOctetstring ::= OCTET STRING
typedef OCTET_STRING MyOctetstring;
MyOID ::= OBJECT IDENTIFIER
typedef OBJECT_IDENTIFIER MyOID;
MyString ::= PrintableString
typedef PrintableString MyString;
MyTime ::= UniversalTime
typedef UniversalTime MyTime;

Notice that the equivalent of the ASN.1 type NULL had to be renamed ASN_NULL in C++, since NULL is a keyword in C and C++. Except for this modification and the underscores used to replace spaces, all names are the same. The only primitive ASN.1 type left out of this pattern is the ENUMERATED type, which maps to the C++ enum.

Types defined to be equivalent to other user-defined types are also translated as typedefs, referring to the appropriate module class if the reference is to another module:

<table>
<thead>
<tr>
<th>ASN.1</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyNewType ::= MyOldType</td>
<td>typedef MyOldType MyNewType;</td>
</tr>
<tr>
<td>ThisModuleType ::= OtherModule::Type</td>
<td>typedef OtherModule::Type ThisModuleType;</td>
</tr>
</tbody>
</table>

As described in section 1.2, for INTEGER and BITSTRING ASN.1 types, specific integers or bit positions can be associated with identifiers, similar to an enumeration in C. Thus, these constructs are implemented as enums. Currently the enumeration is defined anonymously above the INTEGER or BIT_STRING type in the C++ code, rather than as an integral part of the type; in order to indicate the enumerated names' attachment to the appropriate type, the type name is used as a prefix for the enumerated names. This is the only instance where more than one ASN.1 type maps to a C++ type: INTEGER and BITSTRING enumerations as well as the ENUMERATED type itself all become
enums. The mapping from ASN.1 to C++ is shown by the following example of an INTEGER, MyInt, which has an enumeration:

\[
\begin{align*}
\text{ASN.1} & \quad \text{C++} \\
\text{MyInt} & \quad \text{enum} \{ \\
& \quad \text{first}(1), \text{second}(2), \text{third}(3), \text{fourth}(4), \text{fifth}(5), \text{seventh}(7) \} \\
\text{MyInt} & \quad \text{MyInt}\_\text{first} = 1, \\
\text{MyInt} & \quad \text{MyInt}\_\text{second} = 2, \\
\text{MyInt} & \quad \text{MyInt}\_\text{third} = 3, \\
\text{MyInt} & \quad \text{MyInt}\_\text{fourth} = 4, \\
\text{MyInt} & \quad \text{MyInt}\_\text{fifth} = 5, \\
\text{MyInt} & \quad \text{MyInt}\_\text{seventh} = 7 \\
\}; & \quad /* \text{end enum} */ \\
\text{typedef} & \quad \text{INTEGER} \quad \text{MyInt};
\end{align*}
\]

2.2.2 Constructed Types

ASN.1 types SEQUENCE OF and SET OF are translated using the SEQUENCE_OF and SET_OF templates defined in the C++ class hierarchy. For instance,

\[
\begin{align*}
\text{ASN.1} & \quad \text{C++} \\
\text{MyIntSequence} & \quad \text{typedef SEQUENCE_OF<INTEGER> MyIntSequence;} \\
\text{MyIntSet} & \quad \text{typedef SET_OF<INTEGER> MyIntSet;}
\end{align*}
\]

ASN.1 types SEQUENCE and SET, however, become their own classes, inheriting publicly from classes of the same names, with the component type definitions as public data members. For instance,
An assignment operator and a constructor are defined for each SEQUENCE and SET, but otherwise the C++ follows the same pattern and naming as the ASN.1. Notice that component types are kept distinct regardless of repeated names, because they can only be referred to in terms of the parent class, as in "MySet::int_part" versus "MySequence::int_part." Notice also that the component types in the SEQUENCE are defined within a "REQUIRED" template; this is to distinguish them from OPTIONAL and DEFAULTS component types, which are described below. SET component types are always defined within an OPTIONAL template, to reflect the fact that members of a set are, by definition, optional in any given instance.

2.2.2.1 REQUIRED, OPTIONAL, and DEFAULTS

Recall the FTAM type F-ERASE-response, shown in Section 1.2. This type was defined to be a SEQUENCE with two component types: one with a
DEFAULT value and one that was OPTIONAL. The distinction between an OPTIONAL component type and one with a DEFAULT value is that when an OPTIONAL component type's value is left out, that part of the data is omitted, whereas when the value for a component type with a DEFAULT is left out, the data still exists and has the DEFAULT value.

In C++ the distinction between required (the general case), OPTIONAL, and DEFAULT component types is handled by the REQUIRED, OPTIONAL, and DEFAULTS templates defined in the class hierarchy. Any OPTIONAL component types (including all SET components) are defined using the OPTIONAL template; component types with DEFAULT values are defined using the DEFAULTS template; and all other component types are defined using the REQUIRED template. DEFAULT values are assigned to the associated component type. Thus, F-ERASE-response, from FTAM, as shown in Section 1.2.2,

\[
\text{F-ERASE-response} ::= \text{SEQUENCE} \\
\text{action-result Action-Result DEFAULT success,} \\
\text{diagnostic Diagnostic OPTIONAL}
\]

is translated as follows:

```cpp
class F_ERASE_response: public SEQUENCE {
public:
  DEFAULTS<Action_Result> action_result;
  OPTIONAL<Diagnostic> diagnostic;
  void operator= (BaseType& pe) {SEQUENCE::operator= (pe); }
  F_ERASE_response(): action_result (this), diagnostic (this) {
    action_result = 0; /* success */
  }
}; /* end F_ERASE_response */
```

27
The component type "action_result," which was given a DEFAULT value of "success" in the ASN.1, is here defined using the DEFAULTS template, and the integer value of "success" is assigned to "action_result" in the constructor for the SEQUENCE. Diagnostic, which was defined as OPTIONAL in the ASN.1, is here defined using the OPTIONAL template. These templates, and the inclusion of the DEFAULT value as an integral part of the type, preserve component type attributes, as described by ASN.1 specifications, better than any other translator to date.

2.2.2.2 Nested Types and Missing Names

Constructed types, as well as the CHOICE type, are often nested. Furthermore, when these types are nested, names for the component types that are themselves constructed (or CHOICE) types are often left out. In order to keep the definitions compatible with the templates, CATY must undo nesting. Nested constructed types are defined above the parent types, and an intermediate name is invented if necessary. For instance, the following nested ASN.1 type is adapted from FTAM [8]:

```
Charging ::= SEQUENCE OF SEQUENCE {
      resource-identifier   GraphicString,
      charging-unit        GraphicString,
      charging-value       INTEGER
}
```

Since the whole definition of the un-named SEQUENCE above cannot be specified within the "SEQUENCE_OF<Type>" syntax, CATY defines the SEQUENCE above the original type definition and assigns it a name of the form <ParentTypeName>_<ComponentType>, as follows:
class Charging_SEQUENCE: public SEQUENCE {
    public:
    REQUIRED<GraphicString> resource_identifier;
    REQUIRED<GraphicString> charging_unit;
    REQUIRED<Integer> charging_value;

    void operator= (BaseType& pe) { SEQUENCE::operator= (pe); }
    Charging_SEQUENCE () :
    resource_identifier (this),
    charging_unit (this),
    charging_value (this) {} 
}; /* end Charging_SEQUENCE */

typedef SEQUENCE_OF<Charging_SEQUENCE> Charging;

2.2.3 Meta-Structural Types

The meta-structural types are CHOICE and ANY. ANY types are translated the same as primitive types:

<table>
<thead>
<tr>
<th>ASN.1</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyAny ::= ANY</td>
<td>typedef ANY MyAny;</td>
</tr>
</tbody>
</table>

CHOICE types, on the other hand, are translated in a manner similar to that of the SEQUENCE and SET. They inherit from the class CHOICE, and their component types, or alternatives, are defined using the ALT template. Additionally, an enum is generated for the purpose of accessing the appropriate CHOICE element using only its name, by including each enumerated value in the ALT template. Thus, the example "DistinctChoices" from section 1.2.2,

```
DistinctChoices ::= CHOICE {
    first-choice [1] INTEGER,
    second-choice [2] INTEGER,
    third-choice [3] REAL
}
```

would be translated as follows:
class DistinctChoices: public CHOICE {
    public:
    ALT <0, EXPLICIT <CONTEXT, 1, INTEGER>> > first_choice;
    ALT <1, EXPLICIT <CONTEXT, 2, INTEGER>> > second_choice;
    ALT <2, EXPLICIT <CONTEXT, 3, INTEGER>> > third_choice;
    void operator= (BaseType& pe) { CHOICE::operator= (pe); }
    enum {
        type_first_choice = 0,
        type_second_choice = 1,
        type_third_choice = 2
    };
    /* end enum */
    DistinctChoices ():
        first_choice (this),
        second_choice (this),
        third_choice (this) {}
    }
    /* end DistinctChoices */

Notice that the tags specified in the ASN.1 version are not necessary for distinguishing and accessing the appropriate choice in the C++ version, since the enumeration allows access via "type_<choiceElementName>"; however, all the original tag information is preserved in the tag template (EXPLICIT or IMPLICIT). This and other translations of tag information are explained in section 2.3.

2.2.4 Subtypes

Subtypes are not translated by CATY as of this date. Type definitions with subtypes attached are parsed but ignored. It is hoped that subtypes will soon be defined within the C++ class hierarchy, at which time CATY will translate them.

2.3 Tags

Tagged types are translated using the IMPLICIT and EXPLICIT templates defined in the C++ class hierarchy. Which template to use is determined by the standard rule: if not specified, the tag style is the default specified at the beginning of the module, and if a default tag style is absent, use EXPLICIT tags and the EXPLICIT template. A tag can occur anywhere a type can, and so the
IMPLICIT and EXPLICIT templates can occur anywhere a type can: in typedefs, as component types or ALT types, or within other templates such as the OPTIONAL and SET_OF templates. Examples of the use of the EXPLICIT template were shown above in the translation of "DistinctChoices." The template takes three parameters: the tag class, the tag number, and the base type. The IMPLICIT template is used in the same way. In this manner, the tag information is preserved as an integral part of the type it modifies, whereas tags were often left in #define statements by previous translators.
3 The Design and Implementation

CATY is a simple one-pass translator that uses slightly modified versions of the *lex* lexer and the *yacc* parser from the ROSY source code distributed as part of ISO/IEC 9899:1999 [16]. CATY source code is available by anonymous ftp on actor.cs.vt.edu in the directory /pub/CATY. This chapter describes the design of CATY: data structures are described in Section 3.1, and the program is described in Section 3.2.

3.1 Data Structures

Figure 5 shows the subset of the ASN.1 grammar that is translated by CATY. This grammar is based on that used in the parser for ROSY in ISO/IEC 9899:1999 [16]. Notice that *Type* appears in several places in the tree, and that its definition accounts for most of the tree. The first data structure discussed will be the *Type_Element*, which holds information gathered from the *Type* subtree. *EnumList_Type* and *Value_Type*, two important parts of *Type_Element*, will be discussed in the following two sections. Finally, *DefList_Type*, which acts essentially as a symbol table and is not part of the ASN.1 grammar, will be described.

3.1.1 Types

**Definition**

*Type_Element* is the most important data structure in CATY. It holds the information for each type found in the ASN.1 input. The full C definition of *Type_Element* is shown in Figure 6.
enum valtype { NoValue = 0, ValIsNumber, ValIsName };  

typedef struct NumericValue_T {  
    int code;    /* enum valtype */  
    int number;  /* * Numbers can be specified by */  
    char *name;  /* * either an integer or an */  
    char *identifier. */  
} NumericValue_Type;  

typedef struct Tag_T {  
    char *class;       /* A Tag consists of a class, */  
    char *explicit;    /* "IMPLICIT" or "EXPLICIT", */  
    NumericValue_Type val;  /* and a value */  
} Tag_Type;  

typedef struct EnumList_T {  
    NumericValue_Type val;  /* An enumerated element consists */  
    char *enumName;       /* of a value, a name, and a next */  
    struct EnumList_T *next;  /* element to continue the list. */  
} EnumList_Type;  

typedef struct Value_T {  
    int code;           /* Token-value of one of the types */  
    union {  
        int integer;  /* BOOLEAN, INTEGER */  
        double real;  /* REAL */  
        char *string;  /* STRING, DEFINED, ENUMERATED */  
    } val;  
    /* struct Value_T *next; in case we have a list; */  
    not currently used */  
} Value_Type;  

/* Information associated with each type assignment or with each element of an aggregate's type list: */  

typedef struct ELEMENT {  
    char *basetype;    /* the element's type */  
    char *type;        /* the element's type, with tag template */  
    char *ident;       /* identifier name */  
    char *ROD;         /* Required, Optional, or Default */  
    int typenum;       /* token number representing its type */  
    int defined;       /* True if defined type, False otherwise */  
    int implicit;      /* True if implicit tag, False otherwise */  
    Tag_Type *tag;     /* Tag (class + value) */  
    EnumList_Type *tag;    /* enumerated types associated w/ ENUMERATED, */  
    /*enumlist; */  
    Value_Type *defval; /* default value, if Default */  
    struct ELEMENT *next;  /* next type if we are in an aggregate's list */  
    struct ELEMENT *memberlist;  /* elements of aggregate element */  
} Type_Elem;  

Figure 6: Definition of Type_Elem  

34
Instances of type Type_Element are derived by the Type subtree of the tree shown in Figure 5. For example, a very simple ASN.1 TypeAssignment such as

\[
\text{MyInt ::= INTEGER}
\]

would be represented as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>basetype</td>
<td>&quot;INTEGER&quot;</td>
</tr>
<tr>
<td>type</td>
<td>&quot;INTEGER&quot;</td>
</tr>
<tr>
<td>ident</td>
<td>&quot;MyInt&quot;</td>
</tr>
<tr>
<td>typedef</td>
<td>INTEGER</td>
</tr>
<tr>
<td>(constant integer token value returned from lexer)</td>
<td></td>
</tr>
</tbody>
</table>

All other fields would be NULL or zero. If the type is tagged, its tag information is stored in the tag field, and the type field contains the C++ translation of the type name within the tag template. For instance:

\[
\text{MyTaggedInt ::= [APPLICATION 5] INTEGER}
\]

would have the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>basetype</td>
<td>&quot;INTEGER&quot;</td>
</tr>
<tr>
<td>type</td>
<td>&quot;EXPLICIT &lt;APPLICATION, 5, INTEGER&gt;&quot;</td>
</tr>
<tr>
<td>ident</td>
<td>&quot;MyTaggedInt&quot;</td>
</tr>
<tr>
<td>typedef</td>
<td>INTEGER</td>
</tr>
<tr>
<td>implicit</td>
<td>0</td>
</tr>
<tr>
<td>tag-&gt;class</td>
<td>&quot;APPLICATION&quot;</td>
</tr>
<tr>
<td>tag-&gt;plicit</td>
<td>&quot;EXPLICIT&quot;</td>
</tr>
<tr>
<td>tag-&gt;val.code</td>
<td>VallsNumber</td>
</tr>
<tr>
<td>tag-&gt;val.number</td>
<td>5</td>
</tr>
</tbody>
</table>
It is possible, though uncommon, for a tag number to be expressed as an identifier rather than an integer. In such a case, tag->val.code would be ValIsName, and tag->val.name would be a character string holding the identifier.

Constructed and CHOICE types make more thorough use of the Type_Element type. The field memberlist points to the type or list of types from which the outer type is constructed; and within the list, the pointer next points to the next element in the list. Members of a constructed type’s list use the field ROD to indicate “REQUIRED”, “OPTIONAL”, or “DEFAULTS” (or "ALT" for CHOICE elements). If an element of the memberlist is found to be a user-defined type that is not yet defined, the name of this type will be stored in the parent type’s field choke.

For example, the nested type Charging from FTAM was described in chapter 2:

\[
\text{Charging} ::= \text{SEQUENCE OF SEQUENCE} \{ \\
\quad \text{resource-identifier} \quad \text{GraphicString,} \\
\quad \text{charging-unit} \quad \text{GraphicString,} \\
\quad \text{charging-value} \quad \text{INTEGER}
\}
\]

Charging would be stored in the Type_Element type as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>basetype</td>
<td>“SequenceOf &lt;Charging_SEQUENCE&gt;”</td>
</tr>
<tr>
<td>type</td>
<td>“SequenceOf &lt;Charging_SEQUENCE&gt;”</td>
</tr>
<tr>
<td>ident</td>
<td>“Charging”</td>
</tr>
<tr>
<td>typenum</td>
<td>SEQUENCEOF</td>
</tr>
<tr>
<td>memberlist-&gt;</td>
<td>“SEQUENCE”</td>
</tr>
<tr>
<td>basetype</td>
<td>“SEQUENCE”</td>
</tr>
<tr>
<td>type</td>
<td>“Charging_SEQUENCE”</td>
</tr>
<tr>
<td>ident</td>
<td>SEQUENCE</td>
</tr>
<tr>
<td>typenum</td>
<td>“GraphicString”</td>
</tr>
<tr>
<td>memberlist-&gt;</td>
<td>“resource_identifier”</td>
</tr>
<tr>
<td>basetype</td>
<td>“GraphicString”</td>
</tr>
<tr>
<td>type</td>
<td></td>
</tr>
<tr>
<td>ident</td>
<td></td>
</tr>
</tbody>
</table>
Thus, the Type_Element data structure allows unlimited nesting of types.

**Operations**

The operations on instances of Type_Element are new_type, free_type, and enq_type.

*New_type* is called from the *yacc* parser, whenever a BuiltinType or DefinedType is recognized. *New_type* takes as a parameter the token returned from the lexer for the given type. *New_type* allocates space for a new type element, assigns the parameter to the *typenum* field and default values to each of the other fields, and returns a pointer to the new Type_Element.

*Free_type* is called when the type stored in a Type_Element structure has been printed in the output file. *Free_type* frees each pointer field of the Type_Element pointer given as a parameter, then frees the Type_Element itself.

*Enq_type* takes two parameters: a Type_Element and a list (of the same type) to which it is added. *Enq_type* is called when a constructed type’s DEFAULTS elements need to be stored for later use; or when a type is based on
an as-yet undefined user-defined type, and needs to be stored for later reordering. Enq_type adds the Type_Element to the given list.

3.1.2 Enumerations

Definition

EnumList_Type is the part of Type_Element that holds enumerations that are found in some INTEGER and BITSTRING type definitions, and in all ENUMERATED type definitions. These enumerations are derived in the NNIlist subtree of the parse tree shown in Figure 5. An example of such an enumeration was shown in Chapter 2:

```
MyInt ::= INTEGER
{
    first(1), second(2), third(3),
    fourth(4), fifth(5), seventh(7)
}
```

MyInt would be stored as a Type_Element as it was shown in section 3.1.1 above, except that it would use the enumlist field:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enumlist-&gt;</td>
<td></td>
</tr>
<tr>
<td>enumName</td>
<td>“first”</td>
</tr>
<tr>
<td>val.code</td>
<td>VallsNumber</td>
</tr>
<tr>
<td>val.number</td>
<td>1</td>
</tr>
<tr>
<td>next-&gt;</td>
<td></td>
</tr>
<tr>
<td>enumName</td>
<td>“second”</td>
</tr>
<tr>
<td>val.code</td>
<td>VallsNumber</td>
</tr>
<tr>
<td>val.number</td>
<td>2</td>
</tr>
<tr>
<td>next-&gt;</td>
<td></td>
</tr>
<tr>
<td>enumName</td>
<td>“third”</td>
</tr>
<tr>
<td>val.code</td>
<td>VallsNumber</td>
</tr>
<tr>
<td>val.number</td>
<td>3</td>
</tr>
<tr>
<td>next-&gt;</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
...and so on, terminated with NULL after the "seventh" node. As with the val subfield of tag, it is possible, though uncommon, for an enumerated type's number to be expressed as an identifier rather than an integer. In such a case, val.code would be VallsName, and val.name would be a character string holding the identifier.

EnumList_Type is also used for the list enum_deflist, which acts as a symbol table for all enumerations that have been printed in the output file so far. Enum_deflist is consulted whenever a number appears as an identifier rather than as an integer, such as in a tag, a DEFAULTS value, or in another enumeration. The enumName field within this list is prefixed with the identifier of the type in which the enumeration is found, so that, upon lookup, one type's enumeration can be distinguished from another's. For instance, the enumName field of the enum_deflist entry for "second" above would be "MyInt_second".

Operations

The operations on instances of EnumList_Type are do_enum and get_enum.

Do_enum is called right before a type with an enumeration is printed, so that the enumeration will be printed above the type's definition. Do_enum takes three parameters: the enumlist field, the type's identifier, and a boolean telling whether the type is an element of a memberlist. Do_enum prints the enumeration in the output file (prefixing with the given identifier, and indenting if the type is in a memberlist), adds each name in the enumeration to enum_deflist, and frees the enumlist.
Get_enum is called when a numeric value, which is specified by an identifier rather than an integer, is about to be printed. The identifier (prefixed with the appropriate type name) is given as a parameter, and the integer value is returned if it is found in enum_deflist.

3.1.3 Values

Definition

Value_Type is the structure used to store an instance of an ASN.1 Value. Currently a Value is only translated when found as a DEFAULT in a SEQUENCE or SET, and then only when it appears as an integer, a real, or a character string (identifier). Work is in progress to translate Values as parts of SubType specifications. See the yacc grammar in the CATY source code for a more detailed definition of uses and kinds of Values.

An instance of a Value_Type is derived in the Value subtree of the parse tree (not shown in detail in Figure 5), and stored in the field defval in the Type_Element for which the Value is a DEFAULT value. Recall F-ERASE-response from FTAM, discussed in Chapter 1:

```
F-ERASE-response ::= SEQUENCE {
    action-result Action-Result DEFAULT success,
    diagnostic Diagnostic OPTIONAL
}
```

This type definition would be stored as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>basetype</td>
<td>&quot;SEQUENCE&quot;</td>
</tr>
<tr>
<td>type</td>
<td>&quot;SEQUENCE&quot;</td>
</tr>
<tr>
<td>ident</td>
<td>&quot;F_ERASE_response&quot;</td>
</tr>
<tr>
<td>typename</td>
<td>SEQUENCE</td>
</tr>
</tbody>
</table>
...and so on. Notice that the value "success" is coded as an ENUMERATED; this causes the program to call get_enum to find the integer value associated with the DEFAULT. In the future, if the class hierarchy is expanded to handle translations of other kinds of DEFAULT values, it will be easy to have CATY recognize Values other than integers, reals, or strings, by adding new types to the val union within the Value_Type definition and by making use of the next field that is currently commented out of Value_Type.

**Operations**

New_value is the only operation on instances of Value_Type. New_value is similar to new_type: it takes as a parameter an appropriate token name, allocates space for the Value, assigns the given parameter to the code field, and (in comments) sets the next field to NULL. New_value then returns the pointer to the new Value_Type.
3.1.4 Defined List

**Definition**

DefList_Type is not part of the ASN.1 grammar; it is the type of a symbol table. It is defined as follows:

/* List of what types have been defined already in this file: */

typedef struct DEF_LIST_ELEM {
    char *name;  /* the defined type's identifier */
    int typenum;  /* the type's typenum (token number) */
    struct DEF_LIST_ELEM *next;
} DefList_Type;

The only instance of DefList_Type is type_deflist, the symbol table used to keep track of the ASN.1 type assignments that have been translated into the output file so far. Type_deflist is used to check for undefined type names within a constructed or CHOICE type, or to find the original BuiltinType typenum associated with a user-defined type.

**Operations**

The operations on type_deflist are contains_undefined, add_def_type, and get_typenum.

Contains_undefined is called before a type assignment is printed, to determine whether there are forward references. It takes as a parameter an instance of a Type_Element, and checks the definition for as-yet undefined identifiers (i.e., names that are not in type_deflist). Contains_undefined returns the empty string ("") if there are no undefined identifiers, or the name of the first undefined identifier if one is found. (In the calling routine, if contains_undefined returns the empty string, then the type is printed; otherwise, the returned string is
assigned to *choke* and the type element is enqueued (see enq_type) rather than printed.)

Add_def_type is called whenever a type assignment is printed in the output file. It takes as parameters the name and *typenum* of the type being defined, and adds the information to the type_deflist.

Get_typenum is called to determine the *typenum* of a user-defined type, for use in error checking. Get_typenum takes the *basetype* name of the user-defined type, and checks type_deflist to see if the name has been defined. If not, 0 is returned, otherwise, the *typenum* of the defined type is returned.

### 3.2 The Program

The main routine, after processing command-line flags, first calls *yyparse*, which starts the *yacc* parser. Whenever TypeAssignment is reached in the parse tree, the type (Type_Element) found is processed (either printed or enqueued for forward reference resolution) by the routine do_type, which in turn handles nesting by calling do_struct_type and do_offtype as appropriate. These three routines are discussed in the next section. The section after it describes the resolution of forward references after the parsing is done.

### 3.2.1 Important Routines

Three routines, do_type, do_struct_type, and do_offtype, are responsible for printing the translations of each type definition into the output file. These three routines call each other recursively as needed to handle nested types.
3.2.1.1 do_type

Do_type is called whenever a type assignment is encountered in the yacc grammar, or when the forward references, in a type that had to be enqueued earlier, are found to be resolved. Do_type takes as parameters an instance of type Type_Element that has been taken from the TypeAssignment node of the parse tree, and a list, called undeflist, on which types with forward references are enqueued. If the type is user-defined, a constructed type, or a CHOICE, contains_undefined is called to determine if the type contains any forward references. If it does, the name of the undefined type is stored in choke, the type is enqueued on undefined, and do_type returns. Otherwise, the type's identifier is added to the type_deflist, the type field is assigned a translation of the type including the tag template, and the type assignment is printed according to its type. Primitive types, useful types, user-defined types, and ANY are simply printed from within do_type, since their definitions are simple typedefs (with the possible addition of an enumeration, for which do_enum is called). SEQUENCE_OF and SET_OF types are handled the same way, unless they contain nested definitions, in which case do_oftype is called to handle the nested type first. For CHOICE, SEQUENCE, and SET types, do_struct_type is called.

3.2.1.2 do_struct_type

Do_struct_type is called whenever SEQUENCE, SET, or CHOICE types, which become new classes, need to be printed. Do_struct_type takes as parameters the type, basetype, ident, typenum, and memberlist fields of the type to be printed, as well as undeflist. First all the members in the memberlist are preprocessed: names are created if no identifier was specified in the ASN.1
definition, and any member that is a nested constructed type is sent to do_struct_type or do_oftype to be printed. Second, the class definition for the type is printed, including a list of each member in the memberlist (with enumerations printed above members by do_enum if necessary), an assignment operator, an extra enum for identification of CHOICE elements if the type is a CHOICE, and a constructor with assignment statements for any default values.

3.2.1.3 do_oftype

Do_oftype is called whenever the type in a SEQUENCE_OF or SET_OF is a nested constructed or CHOICE type. Do_oftype takes as parameters the same Type_Element that was originally passed to do_type, and the undeflist. First a name is created for the memberlist type if no identifier was specified in the ASN.1 input. The name is recorded in the type field of the parent type, and used in the definition of the member type. Then do_struct_type is called if the member is a SEQUENCE, SET, or CHOICE; or, do_type is called if the member is a SEQUENCE_OF or SET_OF.

3.2.2 Resolving Forwards

After all of the ASN.1 input has been parsed, (if the NOREORDER flag "-n" was not specified,) undeflist is iteratively checked for types that can now be defined, since the user-defined types on which they depended have been previously defined in the output file. All types whose forward references can be resolved, are printed (via do_type). When undeflist does not get any smaller, the remaining types are printed to the output file under the comment "/* possibly
undefined types "/*" and their names and choke fields are given in an error message on stdout.
4 Evaluation

4.1 Speed and Throughput

Performance was measured by comparing the speed and throughput of CATY and PEPSY [16] in translating three well-known ASN.1 specifications: SNMP [5], FTAM [8], and Z39.50. SNMP is the Simple Network Management Protocol, usually used for LAN and TCP/IP network management. Its ASN.1 specification is relatively small, with only 25 type definitions, 316 lines, and 6,091 characters. Z39.50 is the specification for an information retrieval protocol, used to implement WAIS (Wide Area Information Service). Z39.50 is larger, with 79 type definitions, and is densely packed, with 612 lines containing 32,098 characters. FTAM, the largest of the test modules (95 type definitions, 1673 lines, 40,655 characters) defines the protocol for File Transfer, Access and Management, which allows file manipulation on remote systems.

CATY was compared for performance with the ASN.1-C translator PEPSY from ISODE [16]. The two translators were each run 10,000 times on each of the three ASN.1 specifications described above, on a Sun Sparcstation running SunOS 4.1.1. Time was measured in processor clock ticks used by the programs. Throughput was obtained using three measures: number of type definitions per module, lines per module, and characters per module, each divided by the number of clock ticks per module. In all three cases, CATY is, on the average, 27% faster than PEPSY. Figure 7 summarizes the results.
<table>
<thead>
<tr>
<th></th>
<th>PEP#Y</th>
<th>CATY</th>
<th>PEP#Y</th>
<th>CATY</th>
<th>PEP#Y</th>
<th>CATY</th>
<th>PEP#Y</th>
<th>CATY</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SNMP</strong></td>
<td>.28405</td>
<td>.19957</td>
<td>88.01</td>
<td>125.90</td>
<td>1112</td>
<td>1591</td>
<td>21443</td>
<td>30674</td>
</tr>
<tr>
<td><strong>FTAM</strong></td>
<td>1.15455</td>
<td>.93050</td>
<td>82.28</td>
<td>102.10</td>
<td>1449</td>
<td>1798</td>
<td>35213</td>
<td>43692</td>
</tr>
<tr>
<td><strong>Z39.50</strong></td>
<td>.97088</td>
<td>.84412</td>
<td>81.37</td>
<td>93.59</td>
<td>630</td>
<td>725</td>
<td>33061</td>
<td>38025</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>N/A</td>
<td>N/A</td>
<td>83.89</td>
<td>107.20</td>
<td>1064</td>
<td>1371</td>
<td>29906</td>
<td>37464</td>
</tr>
</tbody>
</table>

Figure 7: CATY vs. PEP#Y as tested on three ASN.1 modules, based on processor clock ticks used by each program.
4.2 Correctness of Output

CATY was designed for compatibility with the work of Kafura, Mullins, and Lavender [10, 13]. CATY’s translations of FTAM, Z39.50, and SNMP, described above, were inspected according to the specifications set by Kafura, Mullins, and Lavender. All type definitions and their attributes (component types, tags, and EXPLICIT, IMPLICIT, OPTIONAL, and DEFAULT specifications) were translated (that is, it was verified that nothing was left out), and the translated code was found to be identical to specifications.

In order to ensure that the output code would execute correctly in conjunction with class hierarchy routines, three test files were translated and used as header files for executable programs. One of the test files was the PasswordLookup ASN.1 specification provided by ISODE, for use with a distributed password lookup service. The other two test files were original: Test1 was designed to test varieties of component types of a SEQUENCE, and Test2 tested instances of SET, SEQUENCE, and CHOICE. Each program declared instances of each type output from CATY, assigned values to those types, encoded and pretty-printed the encoded data using routines linked from the class hierarchy. Figure 8 depicts a structure chart showing the relationships between CATY, the ASN.1 specifications, and class hierarchy routines defined by Mullins [13]. Figure 9 shows the types and type attributes tested by each file. Checks in Figure 9 indicate that a file contained a certain type or attribute, and that it was encoded and printed correctly by the corresponding test program. Absence of a check indicates that the type or attribute was not tested by a certain file. All tested types and attributes were found to be encoded correctly. Exhaustive testing of all possible combinations of types, as they are used in conjunction with
the class hierarchy, is left as the responsibility of those who will pursue further work on the class hierarchy.

Figure 8: Translation, linking, and execution of the PasswordLookup ASN.1 specification.
<table>
<thead>
<tr>
<th></th>
<th>PasswordLookup</th>
<th>Test1</th>
<th>Test2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INTEGER</strong>: assigned an integer</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>assigned an enumerated name</td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td><strong>OCTET STRING</strong></td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Character Strings</td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>User-defined</td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td><strong>NULL</strong></td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td><strong>SET</strong>: all components assigned</td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>subset of components assigned</td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td><strong>SEQUENCE:</strong></td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>DEFAULT component</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>assigned explicitly in program</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEFAULT value assigned implicitly when encoded</td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>OPTIONAL component</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>successfully omitted</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPTIONAL component assigned a value (included)</td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td><strong>CHOICE</strong>: with assignment to different alternative types at different times for one instance</td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>Tagged: UNIVERSAL</td>
<td></td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>APPLICATION</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONTEXT</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>PRIVATE</td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>IMPLICIT</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
</tbody>
</table>

Figure 9: Types and type attributes tested with each ASN.1 source file
5 Conclusion

This chapter concludes the report by summarizing the value of the CATY translator in Section 5.1, and by describing future work in Section 5.2.

5.1 The Value of CATY

5.1.1 CATY as One ASN.1 Translator Among Many

Six ASN.1 translators, including CATY, were described in Section 1.3, and compared based on the following criteria: name and type preservation, consistent access, support of modularity and subtypes, forward reference resolution, flexibility of encoding, and generality of use. It was shown that CATY addresses these issues better than any of the other translators. CATY preserves the names and form of the original ASN.1 by translating into C++ types defined in a class hierarchy that embodies the ASN.1 language elements. Templates defined in this class hierarchy also help preserve the access method, by allowing specifications such as OPTIONAL to be declared in templates rather than implicitly changing the type to a pointer as done in other translators. Modularity is supported by translating each module into a class with the defined types as data members, and interdependent modules need not be translated together for cross-module references to work. Type definitions are automatically reordered as needed to avoid forward references. Finally, encoding rules are left as an entirely separate issue to be negotiated at runtime. Although CATY does not yet provide a translation of the subtype specifications defined as a part of ASN.1, as does SNACC [17], work on subtypes is in progress. Furthermore, CATY has been demonstrated to have better throughput than PEPSY, the translator in ISODE.
5.1.2 CATY as a Supplement to OOSI [10]

One advantage of CATY is that it may be used in any environment, along with the corresponding header files defining the C++ class hierarchy. Another advantage is that CATY is a vital supplement to the OOSI project [10, 11]. OOSI is an object-oriented protocol development framework that focuses on the upper layers of the OSI Reference Model. The goal of OOSI is to provide a flexible framework for the development of concurrent object-oriented applications in a multi-protocol environment [10]. CATY and the C++ class hierarchy will be used as key elements of the presentation layer framework of OOSI.

5.2 Future Work

Work is currently in progress to include ASN.1 subtype specifications within the C++ class hierarchy, and to add the translation of subtypes to CATY. This work will render CATY fully qualified as the best ASN.1 translator available according to the evaluation criteria outlined in section 1.3.

Currently an ASN.1 Value is only translated if it is an integer, a real, or a character string (identifier); the C++ class hierarchy is not designed to accept value assignments of other types. If the class hierarchy is expanded to handle assignments of other types of values, CATY will be easily modified by addition of new types to the val union within the Value_Type definition, by use of the next field that is currently commented out of Value_Type, and by a few simple additions to the yacc parser.

Currently, INTEGER and BIT_STRING enumerations are defined anonymously above the INTEGER or BIT_STRING type in the C++ code, rather
than as an integral part of the type; in order to indicate the enumerated names' attachment to the appropriate type, the type name is used as a prefix for the enumerated names. This is the only instance where more than one ASN.1 type maps to a C++ type: INTEGER and BITSTRING enumerations as well as the ENUMERATED type itself all become enums. It would be possible to add these enumerations to the definitions of INTEGER, BIT_STRING, and ENUMERATED in the class hierarchy; this would complete the class hierarchy's goals to avoid name-mangling and to preserve type information.
References


10. Lavender, R.G., Kafura, D.G., and Mullins, R.W. *Programming with ASN.1 Using Polymorphic Types and Type Specialization*, __.


The following is the "man" page for CATY, available along with the CATY distribution by anonymous ftp to actor.cs.vt.edu:

CATY(1) User Commands CATY(1)

NAME
caty - Class-oriented ASN.1 Translator (Yacc-based)

SYNOPSIS

DESCRIPTION
The caty program translates an ASN.1 module into type definitions, based on the C++/ASN.1 class hierarchy, for use with the C++ programming language.

The `-o' switch directs caty to generate the given file as the output file; the default output file is '<modulename>.h'.

The `-e' switch directs caty to generate the given extension for the output file, preserving <modulename> as the base. The default extension is 'h'.

Normally, caty will reorder type definitions in the output file so as to resolve forward references. The `-n' (noreorder) switch directs caty not to reorder the type definitions.

Normally, caty prints the name of each type as it works. The `-s' (silent) switch disables this behavior.

The `-N' (NAMESPACE) switch directs caty to translate each module into a "namespace" instead of a class, and to issue "using" statements instead of "#include" statements for cross-module references.

The `-v' switch directs caty to print the version on stdout.

FILES
module.<ext> type definitions from module (with extension <ext> specified with the -e switch, or default 'h')

SEE ALSO
CATY: An ASN.1 to C++ Translator in support of Distributed, Object-Oriented Applications

AUTHOR
Wendy Long, Virginia Polytechnic Institute & State University

Sun Microsystems Last change: 22 April 1994