Regular Paper

Implementation and Evaluation of ASN.1 Compiler

TORU HASEGAWA*, SHINGO NOMURA* and TOSHIHIKO KATO*

ASN.1 (Abstract Syntax Notation One) is a formal specification language which provides a means to define the structures of transferred data and their external representation in the OSI application protocols. In implementing OSI application protocol software, implementors have to write encoder and decoder programs which convert data in an internal data representation into an external data representation, and vice versa. The size of encoders and decoders is generally large since the application protocols use complicated data structures in order to provide various communication services. In order to reduce the implementation costs of the encoder and decoder, it is a promising subject to develop a software tool generating those programs automatically. We, therefore, have developed an ASN.1 compiler which generates encoders, decoders and internal data types for individual data types in an ASN.1 specification. We have applied the compiler to generate encoders and decoders of several OSI protocols, such as FTAM and MHS protocols. The size of these programs is almost the same as the manually written encoders and decoders. The performance of generated encoders and decoders is more than 800 K bytes per second on 4 MIPS workstation. It ensures that the compiler can be applied to practical software implementation.

1. Introduction

In the progress of OSI (Open Systems Interconnection) standardization, a number of application layer protocols have been standardized. OSI application layer protocols provide various computer communication services, such as electronic mail and file transfer, in a heterogeneous computing environment. In such an environment, computer systems use different ways to represent data internally. The ASCII and EBCDIC character codes and the variety of the length of integers are typical examples. Data transfer among different kinds of computer systems, therefore, requires the introduction of an external data representation in which the transferred data is expressed in order to be understood commonly. The data in internal representations is converted to and from the external data representation at the transmitting and receiving sides respectively.

ISO and CCITT have standardized a specification language, called ASN.1 (Abstract Syntax Notation One) [1, 2], which is currently used to specify the data types of PDUs (protocol data units) of the OSI and ISDN application layer protocols, in order to provide a means to define abstract data structures of transferred data and their external representation in the OSI environment. ASN.1 provides the syntax rules which define abstract data types, and also defines the encoding rules which determine the standard external data representation for the defined data types. For example, an integer is represented by the INTEGER type which does not specify either a value range or a cardinal, but only specifies that its value is an integer. A communication machine, therefore, implements such an abstract INTEGER as its own integer internal representation. Such an integer internal value is converted to a standard external representation according to the ASN.1 encoding rules for it to be transferred to another machine.

In the development of application layer protocol software, it is required to deal skillfully with abstract data types of PDUs specified by ASN.1 and their external representation. First of all, implementors need to design internal data representations corresponding to the ASN.1 defined abstract data types. For example, ASN.1 SEQUENCE type, which specifies an ordered sequence of data values, has to be skillfully represented in a machine because few programming languages provide such kinds of data types. Secondly, they have to implement the encoder and decoder which convert data in an internal data representation into the external data representation, and vice versa. These conversions are called encoding and decoding respectively. Finally, they write an application program which implements an application protocol behavior. The program refers to and modifies the internal representations of ASN.1 to format PDUs and analyze decoded PDUs.

The implementation of an encoder and decoder has a great influence on the productivity of application protocol software. The size of encoders and decoders is generally large because the application protocols use complicated data structures in order to provide various

*KDD R & D Labs. 2-1-15, Ohara, Kamifukuoka-shi, Saitama, 356, Japan.

communication services. For example, the size of manually implemented encoders and decoders for P1 and P2 protocols of MHS [3] is about 13 K lines in the C language [13]. In order to reduce the implementation cost of the encoder and decoder, it is a promising subject to develop a software tool generating these programs automatically. Since the encoding rules of ASN.1 are rigorously defined, it is considered that automatic program generation is suitable for ASN.1 encoder and decoder implementation.

There are two serious requirements for encoders and decoders which are integrated into practical communication systems. The most important one, which can be easily guessed, is high performance. Another one, which has not been explicitly discussed in the literature [6–8], is productivity and readability of application programs. Programs which format PDUs and analyze decoded PDUs heavily refer to and modify internal representations of ASN.1 abstract data types. The design of the internal representations, therefore, has a great influence on their productivity and readability.

There are two approaches for developing such tools. One is to introduce a tool which generates encoders, decoders and internal data types for individual data types in an ASN.1 specification, called an ASN.1 compiler approach [4–6]. The internal data types retain the structures of original ASN.1 data types. The other is a pre-implemented encoder and decoder approach [7, 8], in which a single, pre-implemented encoder and decoder are used for any data types defined by ASN.1. A pre-defined internal data type is used for any ASN.1 data types. The data types used in an application protocol are described by combining pre-defined types.

We have adopted an ASN.1 compiler approach because a pre-implemented encoder and decoder approach can satisfy neither of the above requirements. First of all, pre-implemented encoder and decoder include so many overheads for interpreting data structures which are specific to each application protocol that they are not able to attain high performance. Secondly, it decreases the readability and productivity of application protocol programs because the identical internal data type cannot preserve the structures of original ASN.1 data types.

Our ASN.1 compiler generates encoders and decoders in the C language which is widely used in the communication software implementations. We have applied the compiler to genrate encoders and decoders of several OSI protocols, such as FTAM [9] and MHS protocols. The size of those programs is almost the same as that of manually written encoders and decoders. The performance of generated encoders and decoders is more than 800 K bytes per second on 4 MIPS workstations. It ensures that the compiler can be applied to developing practical software implementation which is supposed to be executed under local area network (LAN) environments.

In this paper, we describe the design and implementation of the ASN.1 compiler and the evaluation of generated encoders and decoders. In chapter 2 the design strategies of the compiler are presented, and in chapter 3 the structure of the programs generated by the compiler is described. In chapter 4, we describe the implementation of the compiler and the evaluation of generated programs.

Table 1 Built-in types.

	type	value
simple types	BOOLEAN	boolean
	INTEGER	integer
	REAL	real
	BIT STRING	bit string
	OCTET STRING	octet string
	IA5STRING	character string
structured types	SEQUENCE	fixed list of ordered component values
	SET	fixed list of unordered component values
	SEQUENCE OF	list of zero or more ordered component values
	SET OF	list of zero or more unordered component values
	CHOICE	one of the component values

```
PersonalRecord ::= SEQUENCE {
        name    IA5STRING,
        age     Age,
        job     IA5STRING  OPTIONAL }
Age ::= INTEGER
```

Fig. 1 An example ASN.1 specification.

2. Design Strategies

2.1 Outlines of ASN.1

ASN.1 is a formal specification language for defining data structures of PDUs in OSI application layer protocols. ASN.1 provides *built-in* types from which new data types are constructed. Table 1 shows the main built-in types. Built-in types are categorized into *simple types* and *structured types*. A simple type is defined by specifying a set of its values, and a structured type is defined by a reference to one or more other types. Figure 1 illustrates an example of an ASN.1 specification which defines the data type of a personal record with three components: a name, an age and a job name.

ASN.1 also has the encoding rules which determine the external data representation, i.e. the data representation used in the communication, of the data types defined by ASN.1. The external data representation consists of three kinds of octets: *identifier octets, length octets* and *contents octets*.

Identifier octets are determined by either a *tag*, a kind of identifier, or several tags. First of all, each ASN.1 built-in type, such as *IA5STRING* and *SEQUENCE*

```
PersonalRecord ::= SEQUENCE {
    name [APPLICATION 0] IMPLICIT IA5STRING,
    age  Age,
    job  [APPLICATION 1] IMPLICIT IA5STRING
         OPTIONAL }
Age ::= INTEGER
```

Fig. 2 A data type defined by using tags.

types, has a tag which identifies its type. For example, IA5STRING has tag *[UNIVERSAL 22]*, and its identifier octets are 16 in hexadecimal notation. Secondly, additional tags can be assigned to a newly defined ASN.1 type as well as an originally assigned tag. For example, if a structured type has the same data type components, each component must have different identifier octets, which is achieved by adding a different tag to the component. This additional tag is added to a data type which is newly defined and a component of a structured type. For example, the new type definition

"Name::=[APPLICATION 0] IA5STRING"

adds tag [APPLICATION 0] to IA5STRING. Although the identifier octets of IA5STRING value are 16 in hexadecimal notation, those of "Name" value are 60 and 16. Figure 2 gives another example of adding tags to components of the structured type defined in Fig. 1. In Fig. 2, tags "[APPLICATION 0] IMPLICIT" and "[APPLICATION 1] IMPLICIT" are added to the components "name" and "job" whose types are IA5STRING, respectively. The identifier octets of the values "hasegawa" and "researcher" in a value {"hasegawa", 29, "researcher"} of the "PersonalRecord" type are 80 and 81 in hexadecimal notation, respectively. Although these two values have the same type, the identifier octets are different. It is, therefore, very important to skillfully deal with tags which are assigned to a type definition for efficient ASN.1 encoding and decoding.

The length octets hold the length of contents octets. Two forms of length octets are provided by ASN.1 encoding rules: the *definite form* and the *indefinite form*. In the definite form, the length octets specify the length explicitly. Length octets in the indefinite form do not specify the length of contents octets, but a delimiter is used to detect the end of contents octets. The delimiter is called *end-of-contents* octets, which consist of two octets (00 00 in hexadecimal notation).

The contents octets include a data value. For example, in Fig. 2, the contents octets of the value "PersonalRecord" are composed from the encoded octets of component values, "hasegawa", 29 and "researcher".

2.2 Design Principles

The following requirements must be satisfied by a software tool which automatically implements ASN.1 encoder and decoder programs.

—High performance encoding and decoding

—Productivity and readability of application programs

—Abundant error check facilities of decoders

—Supporting of all the ASN.1 data types, including those defined recursively

—Supporting of all the encoding rules

—Portability of generated programs for various operating systems

Of the above requirements, the first two are the most important for realizing an effective encoder and decoder which is used for practical communication system development. In order to satisfy these two requirements, we have taken the ASN.1 compiler approach. We will discuss the reasons for which we have taken the compiler approach and the design principles for fulfilling the requirements after briefly explaining both approaches.

An ASN.1 compiler [4–6] is a software tool which generates encoders and decoders for individual data types in an ASN.1 specification. It also translates each ASN.1 type into a data type of a target programming language. For example, INTEGER of ASN.1 may be translated into int in the C language. On the other hand, pre-implemented encoder and decoder programs perform encoding and decoding for any ASN.1 data types [7, 8]. Generally, all the ASN.1 data types are translated into an identical data type which defines a node of a tree representing the structure of an ASN.1 data type [7, 8]. They refer to trees which consist of these nodes while encoding and decoding. The tree structures are generated from an ASN.1 specification by an ASN.1 pre-compiler.

2.2.1 High Performance of Encoding and Decoding

Since high performance is the most important requirement for an ASN.1 encoder and decoder, the ASN.1 compiler approach is more hopeful. A pre-implemented encoder and decoder is considered not to be able to attain as high performance as an individually generated encoder and decoder because it includes overheads caused by its interpreting the trees in the encoding and decoding. As for the program size of encoder and decoder, however, that of the pre-implemented ones is considered to be smaller than the total size of these generated by an ASN.1 compiler, which was also described in [8].

We concluded that the program size in the ASN.1 compiler approach is not a disadvantage because the generated encoders and decoders may be as large as manually written encoders and decoders, even though they are larger than pre-implemented ones. We also decided that the throughput of encoding and decoding is more important than the size of encoders and decoders.

In addition to adopting the compiler approach, we have adopted the following design principles to implement as effiicient encoder and decoder programs as possible.

(1) Skillful Processing of Tags

It is very important for encoders and decoders to deal with assigned tags for efficient encoding and decoding,

which has not been explicitly stated in the literature [6–8], because encoding and decoding of identifier octets might be heavier than those of length and contents octets which are easily computed. For example, in decoding, all identifier octets must be analyzed so as to check whether they are conforming to the assigned tags. We, therefore, have adopted the following design principles for efficient processing of tags:

—An ASN.1 compiler translates the assigned tags so that generated programs maintain all the possible combinations of assigned tags. Concretely, all the combinations of the assigned tags are converted into identifier octets, which are maintained as constants in a generated program. It releases generated encoders and decoders from computation about assigned tags.

—Tags are implemented compactly. Since the above generated constants are frequently referred to in both encoding and decoding, they are generated as an array of computed identifier octets which makes it easy for encoders and decoders to access them.

(2) 2 Phase Encoding

The structure and length of length octets cannot be determined for structured types until the length of all the components are determined. For example, the length octets whose contents are 32 and 256 octets long are 1 octet and 2 octets long, respectively. It is impossible for an encoder to pack serially all the encoded octets into a result octet string, which means that an 1 phase encoder might require wasteful octet data copy if a length octets' length is longer than expected. 2 phase encoding, therefore, has been adopted for efficient encoding, which has also been adopted by another ASN.1 compiler [6]. In order to pack serially an encoded octet into a result octet string, encoders compute lengths of all the length octets in the first phase, and pack the encoded octets in the second phase.

2.2.2 Readability and Productivity of Application Programs

Readability and productivity of application programs are another important requirement which has not been explicitly stated in the literature [6–8]. Translation of ASN.1 data types into target programming language data types has a great influence on readability and productivity. Despite the importance, ASN.1 pre-compilers [7, 8], used in the pre-implemented encoder and decoder approach, cannot translate ASN.1 data types so that application programs are made easy to read. The fact that ASN.1 pre-compilers translate all the ASN.1 data types into an identical data type of a programming language generally has grave disadvantages.

First of all, this clearly degrades the readability of the program. For example, variables in an application program have the same data type even though they correspond to different ASN.1 types. A reader cannot understand what ASN.1 type the variable is corresponding to. Secondly, a programmer must manipulate variables corresponding to different ASN.1 types in the same way, which may reduce the productivity of application programs. Furthermore, some errors of application programs may not be detected by a target programming language compiler but only detected at the time of execution. For example, suppose a programmer writes codes which assign an integer value to the first component of a variable corresponding to the "PersonalRecord" type in Fig. 2. A target language compiler may not detect this error because there is no way for the compiler to know that the first component is not an INTEGER type but a IA5STRING type.

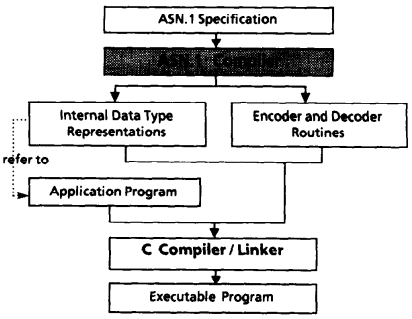

Fig. 3 Implementation procedure.

On the other hand, an ASN.1 compiler, which translates all the ASN.1 types individually, can easily avoid the above disadvantages. This one-to-one translation has also been adopted by another ASN.1 compiler [6]. In order to make translated types as readable as possible, our ASN.1 compiler tries to retain the structures and literal names of original ASN.1 types as much as possible in translating ASN.1 types. This makes application programs easy to write and read.

2.3 Implementation Procedures

We have chosen the C language as a target programming language because it is widely used for communication software implementation. The implementation procedure of application protocol software using our ASN.1 compiler is illustrated in Fig. 3. First of all, from an ASN.1 specification, the ASN.1 compiler translates ASN.1 defined data types into C data types, and generates encoders and decoders in the C language. Secondly, implementors write a protocol behavior part of an application protocol program, which is using and referring to the generated C data types. Finally, all the programs, both automatically generated and manually written, are compiled and linked to executable programs by C language compilers and linkers.

3. Translation and Generation

In this chapter, we describe the details of translation techniques of ASN.1 data types and generation techniques of encoders and decoders. We have designed the generated data types and programs so as to achieve the

Table 2 Translation of simple types.

type	data structures in C language
BOOLEAN	int
INTEGER	int
REAL	double
BIT STRING	defined by struct "STRING"
OCTET STRING	defined by struct "STRING"
IA5STRING	defined by struct "STRING"

requirements, such as portability and readability, described in section 2.2.

3.1 Data Type Translations

3.1.1 Simple Types

All simple types are translated into C data types. Main translations are listed in Table 2.

These translations, except for the STRING type, are achieved by direct mapping. For example, INTEGER and BOOLEAN of ASN.1 are translated into *int* in the C language. REAL is translated into *double* in C.

A string of octets or characters, such as OCTET STRING and IA5STRING, is not directly mapped into a pointer type of char in the C language, but is translated into a C *struct STRING* as illustrated in Fig. 4 because a string of octets does not have a delimiter which indicates the end of a string. This struct has the two members: "*size*" indicating the length of octets or characters, and "*str*" pointing to octets or characters.

The ASN.1 compiler generates a C data type definition from an ASN.1 simple type definition using a typedef statement (see Fig. 4).

3.1.2 Structured Types

(1) SEQUENCE and SET

Each new data type defined by SEQUENCE and SET, which has a similar structure to record type, is translated directly into a C struct whose members correspond to its components. Figure 5 illustrates an example translation of SEQUENCE type. The struct retains a structure and literal names of an original SEQUENCE type. The literal name of a struct type is that of an original ASN.1 type, and the name of each member is also a *reference* name of an original component. This retention also makes it easy for a programmer to declare and manipulate a variable corresponding to a structured type. An optional component, however, whose value's existence is not mandatory, cannot be mapped directly into a component of a translated struct. Even if this optional component were mapped to a single component of the translated struct, the component could not hold both pieces of information: a content value and an indicator which indicates whether its value exists or not. An optional component, therefore, is mapped into two components for holding a value and the indicator. "*job_flag*" in Fig. 5 is an example of the indicator.

```
(1) Data type definitions in ASN.1
    Age ::= INTEGER
    Float ::= REAL
    Name ::= IA5STRING
(2) C data types
    typedef int INTEGER;
    typedef double REAL;
    typedef struct string {
        int size;              /* the length of octets */
        unsigned char *str;    /* the pointer to octets */
        } STRING;
```

Fig. 4 C data types for ASN.1 simple types.

```
(1) Data type definition in ASN.1
    PersonalRecord ::= SEQUENCE {
        name        IA5STRING,
        age         Age,
        job         IA5STRING  OPTIONAL }
(2) Generated codes by the ASN.1 compiler
    typedef struct seq_PersonalRecord {
        IA5STRING   *name;
        Age         age;
        IA5STRING   *job;
        int         job_flag;
        int         esize; /* used by encoder routines */
    } PersonalRecord;
```

Fig. 5 Translation of SEQUENCE type.

```
(1) C types for SEQUENCE OF and SET OF types
    typedef struct lst { /* the node */
        LST     next; /* the pointer to the next node*/
        char    *elem;  /* the pointer to the component */
        } LST;
    typedef struct  seqof {
        LST *first;   /* the pointer to the head of singly-linked list*/
        LST *last;    /* the pointer to the tail of singly-linked list*/
        int size;  /* the number of components */
        int esize;  /* used by encoder routines */ } SEQOF ;
(2) Data type definition in ASN.1
    PersonalRecordList ::= SEQUENCE OF PersonalRecord
(3) Generated codes by the ASN.1 compiler
    typedef SEQOF PersonalRecordList;
```

Fig. 6 Translation of SEQUENCE OF and SET OF types.

(2) SEQUENCE OF and SET OF

It is reasonable for SEQUENCE OF and SET OF, which are ordered and unordered component values, to be represented as either a sequential list or an array; however, an array is not appropriate because the number of components is indefinite. These types are represented as a singly linked sequential list of their components. They are implemented by use of the two data structures in the C language: struct "*SEQOF*" and struct "*LST*" in Fig. 6, because the C language has no type which directly represents a list. Struct "LST" represents a node of the list. It has two pointers: "*elem*"to a component, and "*next*" to the next node. Struct "SEQOF" is a header of the list. It has three members: "*first*" and "*last*" indicating the head and tail of the list, and "*size*" indicating the number of components. The ASN.1 compiler translates SEQUENCE OF and SET OF types into the type "SEQOF" (see Fig. 6).

(3) CHOICE

The CHOICE value holds one of the values out of

```
(1) Data type definition in ASN.1
   PDU ::= CHOICE {
      req ReqPDU, ind IndPDU,
      conf ConfPDU, resp RespPDU }
(2) Generated C type by the compiler
   typedef struct C_PDU {
      int    nth;  /* the indicator of the selected component */
      int    esize; /* used by encoder routines */
      union  {
         ReqPDU *req;  IndPDU *ind;
         ConfPDU *conf; RespPDU *resp ; } elem;
   } PDU;
```

Fig. 7 Translation of a CHOICE type.

possible components. For example, the value of "PDU" type in Fig. 7 holds one of the values whose types are "ReqPDU", "IndPDU", "ConfPDU" and "RespPDU". This type, which is quite similar to union in the C language, cannot be directly translated into union because of the similar reason that an optional component is not directly tanslated into a single component of the translated struct. If it were translated into a union, a programmer could not know to which component type a value is corresponding. It is translated into a C struct consisting of both a component which holds a value and an indicator. Concretely, the struct has the following members: "*nth*" as the indicator for the selected component, and "*elem*" which is a *union* of all the components. Figure 7 illustrates an example translation of the CHOICE type. The ASN.1 compiler generates the struct from a data type defined by the CHOICE type.

3.2 Generation of Encoders and Decoders

3.2.1 Generation Scheme

In order to achieve the criteria described in section 2.2, the ASN.1 compiler generates the encoder and decoder routines in the C language according to the following schemes.

(1) Program Structures

Encoder and decoder routines for simple types are implemented in the encoder/decoder library beforehand. The library also includes primitive routines commonly used by the encoder and decoder routines. Table 3 lists the main routines in the library.

The encoder and decoder routines for structured ASN.1 types, on the contrary, are composed of those for the component types. For example, an encoder routine for the PersonalRecord type calls the encoder routines for the IA5STRING and INTEGER types. The ASN.1 compiler generates encoder and decoder routines for structured types. These routines use the routines in the encoder/decoder library.

(2) Processing of Tags

The ASN.1 compiler translates the tags that are assigned to either a newly defined type or a component type of a structured type into encoded identifier octets. These identifier octets are implemented as an array of octets, named *id_array* (see Fig. 8), which is an argument of encoder and decoder routines. Encoder and decoder routines can easily manipulate identifier octets by directly accessing the id_array. For example, a decoder routine only has to compare identifier octets of the target decoding octets with a corresponding id_array for decoding of identifier octets. This simple structure also makes it easy for encoders and decoders to access the array.

(3) Portability

In order to increase portability, we have designed all the codes of generated and library routines. First, all the routines in the library call only the routines malloc and free which are popularly provided by the C standard libraries. All the generated routines call only the routines in the encoder/decoder library. This makes all the routines free from the differences among various C standard libraries. Secondly, the generated routines and

Table 3 Primitive routines in the library.

name	task
enc_idlen	converting tags into identifier and length octets
skip_idlen	checking the identifier and length octets
checkeoc	checking the end-of-contents octets
e_int	encoding of a integer value
e_string	encoding of STRING value, such as OCTET STRING and IA5STRING value
z_int	calculating the length of a integer value
z_string	calculating of the length of a STRING value, such as OCTET STRING and IA5STRING value
d_int	decoding of a integer value
d_string	decoding of a STRING value, such as OCTET STRING and IA5STRING value
asn_alloc	allocating of memory pool (buffer space)
asn_free	deallocating of memory pool (buffer space)
asn_get	allocating of memory space from memory pool (buffer space)

```
/* Structure of each tag */
typedef struct cell {
    int idsize ; /* the length of encoded octets */
    unsigned char idelem[4] ; /* identifier octets of tags */
    } id_cell ;
/* Structure of id_array */
typedef struct idarray {
    int idlevel ; /* the number of tags*/
    id_cell id[maxtag]; /* an array of tags */} id_array ;
```

Fig. 8 Structure of id_array.

the library also encapsulate the machine dependency of the data representation such as the variety of the byte order of integers and the real representation. For example, in order to ignore the difference of the byte ordering, the routines extract the least significant byte of integer I by calculating I%256. The difference of the representation of double is encapsulated in the encoder and decoder routines for double type.

(4) Simple Memory Management Policy

Since an internal representation of a structured type has pointers to its components, it is necessary to allocate the memory space for the components in the encoding and decoding of PDUs including structured types. The allocated memory spaces need to be deallocated after the encoding and decoding. In order to implement those memory allocations and deallocations efficiently, we have adopted a simple memory management policy according to the following procedures which is also proposed in [6].

—Before the encoding and decoding, a user program allocates a memory space, called *buffer space*, which is large enough for all the components. The buffer space is allocated by the C standard library routine *malloc*.

—When a decoder routine and a user program need to allocate a space for a component, they take the necessary space from the buffer space by calling the encoder/decoder library routine *asn_get*.

—When all the allocated spaces become unnecessary, the buffer space is deallocated by the C standard library routine *free*.

Because the time consuming routines malloc and free are called only once respectively in the encoding and decoding of a PDU, and the routine asn_get is much faster than the routine malloc, the efficient memory management is achieved.

3.2.2 Encoder

The ASN.1 compiler generates two kinds of C routines corresponding to the two phases for a structured type, which are *z_typename* and *e_typename* for the structured type *typename*. Routine z_typename examines the length of a value of the type typename and saves it in the member esize of the translated struct (see Figs. 5, 6 and 7). Routine e_typename converts a value into an octet string. The routine deals with identifier octets using an id_array, and determines the length octets from the saved length. After that, it calls e_routines of the component types in order to encode the components. Figure 9 illustrates routine *e_PersonalRecord* which encodes the PersonalRecord type. The type of encoded octets is the STRING type which represents the ASN.1 OCTET STRING type.

```
e_PersonalRecord ( var, edata, index, ida )
 char    *var; /* the variable for encoding : PersonalRecord type */
 STRING *edata; /* encoded octets */
 int     *index;   /* the starting position of the encoded octet s */
 id_array *ida; /* id_array */
{
  PersonalRecord *p_var;

 p_var  = (PersonalRecord *) var;
 enc_idlen(edata, index, (*p_var).esize, ida);
  /* encoding of the identifier and length octets */
 e_string( (*p_var).name, edata, index,
     id_info + 26 ); /* encoding of name */
 e_int ( (*p_var).age, edata, index,
     id_info + 1 ); /* encoding of age */
 if ( (*p_var).job_flag  = = TRUE)
     e_string( (*p_var).job, edata, index,
         id_info + 27 ); /* encoding of job */
 (*edata).size  =  *index;
}
```

Fig. 9 Encoder routine for PersonalRecord.

3.2.3 Decoder

A decoder routine performs decoding according to the following procedures.

—It checks identifier octets using the id_array of the expected ASN.1 data type.

—It analyzes the length octets. If the length octets are encoded in the definite form, it calculates the length of contents octets. If the length octets are in the indefinite form, the decoder remembers it.

—It allocates a memory space holding the decoded result value.

—If it is decoding a simple type value, it saves the contents octets in the allocated space. In the case of a structured type value, it calls the decoder routines for the components.

—Finally, it checks the existence of the end-of-contents octets if the length octets are in the indefinite form.

If any encoding errors are detected, the decoder routine quits decoding and returns with error information including the error type and the position at which the error is detected.

The error checking facility, whose importance has not been explicitly described in the literature [6–8], is very important because an application program must deal with protocol errors obeying a standard protocol specification. It is desirable for decoders to detect error information as precisely as possible. We decided to make decoder routines perform the following error checks after looking into protocol error manipulation parts of various OSI application protocol specifications. These checks may enable them to write a program so that they implement all kinds of protocol error manipulations. The error checks are the following:

```
d_PersonalRecordList ( var, edata, buf, index, error, ida)
 PersonalRecordList **var; /* the decode result*/
 STRING *edata; /* the decoded octets */
 Buffer *buf; /* the buffer space*/
 int *index; /* a starting point of decoded octets */
 ERROR *error; /* the error information */
   id_array *ida; /* the id_array */
{ LST *wkp; /* the node for a component */
 int nextobj; /* the length of contests octets */
 int cnt; /* the number of the length octets
          according to the indefinite form */
 PersonalRecordList *q_var; /* temporary variable */

 skip_idlen( edata, index, &nextobj, ida, error);
      /* checking the identifier octets and length octets.
         nextobj returns the length of contents octets */
 if ((*error).errlevel<2) {
      /* No error has been occurred in skip_idlen routine ? */
  if ((*var = asn_get(buf, sizeof(PersonalRecordList)))
      = = FALSE) { ERRSET ( FATAL, 6, ida, *index) }
  else {
   q_var = (PersonalRecordList *) (*var);
   for (wkp = (*q_var).first;
        ((cnt = = 0) && (*index < next_obj) || (cnt! = 0)) ;
        wkp = (*wkp).next )
   {
    if ((*wkp) = (LST *) asn_get(buf, sizeof(LST)))
          = = FALSE)
     { ERRSET ( FATAL, 6, ida, *index) }
     else
     {
        (*wkp).next = NILL;
        d_PersonalRecord ( &((*wkp).elem), edata,
          index, error, id_info + 28);
        if ( (*error).errlevel > = 2) { break; }
        else { (*q_var).size + + ; } } }
   if (((*error).errlevel <2) &&
     (((cnt = = 0)&& (*index ! = next_obj)) ||
     ((cnt! = 0) && checkeoc(edata,index,cnt)))) {
      /* the length of contents octets are correct ? */
      ERRSET (FATAL, 2, ida, *index) }
} } }
```

Fig. 10 Decoder routine for PersonalRecordList.

—check for whether identifier octets are correctly encoded or not;

—check for whether contents octets specified by length octets are included or not;

—check for whether mandatory components of SEQUENCE and SET types are included or not; and

—check for whether a memory space for a result value can be successfully allocated or not.

Figure 10 illustrates the decoder routine for PersonalRecordList type defined by the SEQUENCE OF type.

4. Implementation and Evaluation

In this chapter, we describe the implementation details of the ASN.1 compiler and the evaluation of the generated encoders and decoders.

4.1 Implementation

The ASN.1 compiler reads *module definitions*, which consist of ASN.1 data type definitions, for a target application layer protocol. After the reading, it generates the following files from each module definition: a source file including encoder and decoder routines, and a header file including data type definitions in the C language. It also generates the following files: a source file *commdl.c* including the id_arrays, and a header file *asn1toc.h* which is included by an application program. Figure 11 illustrates the structure of the compiler. The primitive routines listed in Table 3 and the C data type definitions for ASN.1 simple types are implemented in the encoder/decoder library program: a source file *asn1lib.c* and a header file *asn1lib.h*.

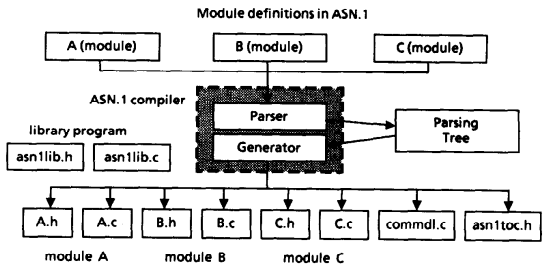

Fig. 11 Structures of the ASN.1 compiler.

The compiler consists of a parser and a generator. The sizes of both parts are about 1.0 k lines and 6.3 k lines long. The parser has been implemented using yacc/lex for a rapid implementation. The parser generates parsing trees from ASN.1 specifications, and also reports error information for ASN.1 specification errors, such as double type definitions and undefined types, and the line number in which the error is detected. The generator generates programs from the parsing tree.

The compiler is implemented in the C language on UNIX and VAX/VMS operating systems. It can generate encoder and decoder routines which can be executed under the UNIX, VAX/VMS and MS-DOS operating systems. Currently, the generated encoder and decoder routines are executed on the following computers: SUN-3, SUN-4, Sony News workstation (UNIX), VAXen (VMS) and PC-9800 personal computers (MS-DOS). The generated programs can be executed on the various computers because of the development described in section 3.2.1.

The compiler does not deal with *macro definitions*, in which a user can define new syntax rules for ASN.1 type definitions, because macro definitions define only the syntax rules, and not the semantics of the definition.

4.2 Evaluations

We have evaluated the performances and sizes of the encoders and decoders generated by the implemented compiler.

4.2.1 Performance of Encoders and Decoders

We measured the performances of the generated encoders and decoders. The following experiments were carried out on a SUN-3 280 workstation (4 MIPS) with

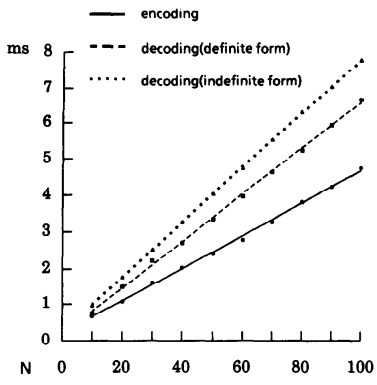

Fig. 12 Encoding/decoding time (1).

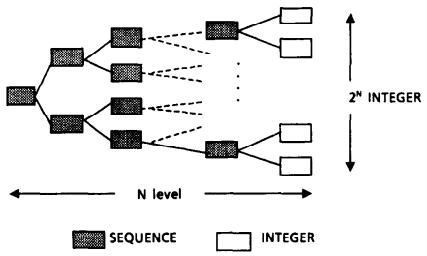

Fig. 13 Test data of the binary tree.

8 mega bytes of memory. Encoding performances using the definite form and those of decoding using both the definite and indefinite forms were measured.

(1) Measurement for structured types

At first, we measured the performances for the complicated values which consist of a number of structured types. Two types of data, sequential lists and binary trees, have been adopted as examples.

The ASN.1 specification of the sequential lists is

```
"List::=SEQUENCE {
  a [1] IMPLICIT INTEGER OPTIONAL,
  b [2] IMPLICIT List OPTIONAL}".
```

Figure 12 shows the results of the experiments for sequential lists, including the encoding performances by the definite form and decoding ones by the definite and indefinite forms. "*N*" in Fig. 12 means the length of the lists.

The structure of binary trees is illustrated in Fig. 13. Fig. 14 shows the results of the experiments for binary trees. "*N*" means the level of a binary tree. If "*N*" equals 8, the tree consists of 2^8 (256) INTEGERs and 2^8-1(255) SEQUENCEs.

The experiments for structured types have shown the following results.

—The generated encoders and decoders are considered to be faster than those of pre-implemented encoder and decoder [7]. It is not easy to compare our implementation with others [7, 8] whose experiments are conducted on different machines and operating systems; however, we compared processing time of ours and pre-implemented one [7] which were measured under similar conditions such that 100 SEQUENCE values were encoded and decoded on about 4 MIPS workstations. Those of our implementation were about 4.8 and 6.5 milliseconds, however, those of pre-implemented one were over 17 and 25 milliseconds. The experiments of another kinds of data, a binary tree, also show a similar result.

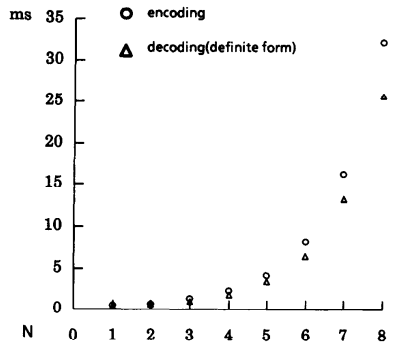

Fig. 14 Encoding/decoding time (2).

—The encoding and decoding time of structured types is in proportion to the number of component structured values.

—Generally, the decoding time is a little larger than the encoding time because decoders have to check identifier octets. Even if an optional component value does not exist in a structured value, decoders have to check the identifier octets in order to confirm that the component value does not exist. The encoding and decoding times of the second experiments are almost the same, since the SEQUENCE type has no optional component.

—The decoding time of indefinite form values is larger than that of definite form values because of the overhead of the function calls which checks the *end-of-contents octets*.

(2) Measurement for Actual PDUs used in OSI protocols

Secondly, we have measured the performances for the PDUs which are transferred in the OSI protocols, FTAM and MHS P2. Table 4 shows the encoding and decoding time, and the structure and the size of the PDUs. Generally, the PDUs defined in actual OSI protocols consist of less than scores of structured and simple type values, and therefore can be encoded and decoded in a shorter time than the complicated values defined in (1).

Among the values in Table 4, IM-UAPDU and File-Contents-Data-Element are used in the data transfer phase. Performances for these PDUs greatly affect the throughput of the mail and file transfer. The encoding time and the decoding time of an IM-UAPDU convey-

Table 4 Encoding/decoding time (3).

PDU	encoding (ms)	decoding (ms)	the number of structured values	the number of simple values	PDU size (bytes)
F-INITIALIZE-Request	1.54	1.32	7	11	64
F-SELECT-Request	0.61	0.59	4	3	17
F-OPEN-Request	0.63	0.57	2	2	16
File-Contents-Data-Element	1.22	0.57	1	2	4,023
IM-UAPDU	2.46	2.59	19	17	2,184

Table 5 Generated programs.

protocol	specification	source program	executable program
FTAM	410 lines	11,560 lines	156Kbytes
MHS P1, P2	290 lines	11,170 lines	143Kbytes

```
#include "asn1toc.h"
main() {
 Buffer *buf; /* the buffer space */
 PersonalRecord *record; ···①
 OCTETSTRING *encdata; /* the encoded octets */
 asn_alloc(buf); /* allocation of the buffer space */
 record = (PersonalRecord *) asn_get(
            sizeof(PersonalRecord));
 record->name =
    (IA5STRING *) asn_get(sizeof(IA5STRING));
 record->name->size = 8; ···②
 record->name->str = "Hasegawa";
  ...
 encode(record, edata, ID_PersonalRecord);
  /* User interface of the encoder routines */
 asn_free(buf); /* deallocation of the buffer space */
  ...
}
```

Fig. 15 An example of application program.

ing a 2 K byte mail content are 2.46 milliseconds (ms) and 2.59 ms respectively. The encoding time and the decoding time of File-Contents-Data-Element conveying a 4 K byte file content are 1.22 ms and 0.57 ms respectively. These results show that the throughput of the generated encoders and decoders for PDUs which have 2 to 4 K byte data is more than 800 K bytes per second on 4 MIPS workstations. This throughput value is much larger than 8 K bytes per second (64 K bps) which wide area network (WAN) provides, and is almost the same as 1.25 Mega bytes per second (10 M bps) which Ethernet provides. Moreover, the existing OSI FTAM implementations [11, 12] on standard workstations, which are about 4 to 6 MIPS, have achieved about ten K bytes per second file transfer under local area network (LAN) environments. This FTAM throughput value is much less than 800 K bytes per second. These ensure that the throughput which the encoders and decoders achieve is fast enough for those to be executed under such environments as WAN and Ethernet. On the other hand, all the PDUs of FTAM protocols, except File-Contents-Data-Element, are used for the communication control. These PDUs can be encoded and decoded within only 1.6 ms by the generated encoders and decoders.

4.2.2 Size of Encoders and Decoders

In order to evaluate the size of generated programs, we have applied the compiler to generate the encoders and decoders for OSI FTAM and MHS P1 and P2 protocols. The results are shown in Table 5. The executable programs are obtained by the VAX/VMS C compiler. The generated programs are considered to be compact, and the sizes are almost as the same as the those of the programs implemented manually. For example, the size of manually implemented MHS P1 and P2 protocol encoders and decoders is about 13 K lines in the C language [13]; however, ours is about 11.2 K lines. Although both programs may have different functions, the similarity ensures that at least the ASN.1 compiler does not generate larger programs than manually implemented ones.

4.2.3 Readability of Application Programs

The one-to-one translation of a data type in ASN.1 into a C type increases the readability and productivity of application programs. Figure 15 illustrates a part of an application program which makes a value of the "*PersonalRecord*" type and encodes it. The advantages are summarized as follows:

—Variables are declared using the literal names of data types in ASN.1 (Fig. 15①).

—In referring to the components of the structured type variables, implementors can use the reference names which are specified in the original ASN.1 specifications (Fig. 15②).

—Since the compiler translates all the component types, except for data types translated into C *int*, into pointers to the component types, implementors only use the-> operator in order to refer to the components.

5. Conclusion and Future Research

In this paper, we have described the implementaion and evaluation of the ASN.1 compiler, which generates the internal representation of ASN.1 data types and the encoder and decoder routines in the C language. Since the ASN.1 compiler generates the encoder and decoder routines for each ASN.1 data type, the generated routines can achieve encoding and decoding efficient enough for practical application protocol software development. The size of the generated programs is almost the same as that of manually implemented programs. The one-to-one translation of an ASN.1 data type into a C type increases the readbility and productivity of application protocol programs because the internal data types retain the structures of original ASN.1 data types. The compiler reduces the implementation costs of the OSI application protocol software drastically.

In order to improve the compiler, the following research is under consideration.

- *Supports for macro notation in ASN.1*
  The compiler is not yet able to deal with *macro notations*, which are used by specifying OSI ROSE protocols [10]. We are going to be implementing the facilities which deal with macro notations defined in OSI ROSE protocols.
- *Integration of ASN.1 compiler and SDL compiler*
  We also have been developing the SDL-C compiler for automatic communication software implementation, and we are planning to integrate an SDL compiler and the ASN.1 compiler.

Acknowledgement

We wish to thank, Dr. K. Ono, Director for his kind support for this study, and also appreciations to Dr. Y. Urano, Deputy director of KDD R & D Labs., Mr. K. Konishi, Group Leader of Communication Software Group and Dr. K. Suzuki, Group Leader of OSI Systems Group, for their helpful suggestions.

References

1. CCITT, Recommendation X.208, Specification of Abstract Syntax Notation One (ASN.1) (Nov. 1987).
2. CCITT, Recommendation X.209, Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1) (Nov. 1987).
3. CCITT Recommendations X.400-X.430 (Oct. 1984).
4. HASEGAWA, T. et al. Automatic Ada Program Generation from Protocol Specifications based on Estelle and ASN.1, *Proc. of the Ninth Int. Conf. Comput. Comm.* (Oct. 1988), 181–185.
5. HASEGAWA, T. et al. Development of Software Tools for ASN.1-Compiler and Editor, *Technical Report DSP 39-4, IPSJ* (Sept. 1988) (in Japanese).
6. NEUFELD, G. W., YANG, Y. The Design and Implementation of an ASN.1-C Compiler, *IEEE Trans. Softw. Eng.*, **16**, 10 (Oct. 1990).
7. NAKAKAWAJI, T. et al. Development and Evaluation of Apricot (Tools for Abstract Syntax Notation One), *Proc. of The Second International Symposium on Interoperable Information Systems* (Nov. 1988), 55–62.
8. OHARA, Y. et al. ASN.1 tools for Semi-automatic Implementation of OSI Application Layer Protocols, *Proc. of The Second International Symposium on Interoperable Information Systems* (Nov. 1988), 63–70.
9. ISO, IS-8571, Information Processing Systems-File Transfer Access and Management (1987).
10. CCITT Recommendations X.219, X.229 (1987).
11. OBANA, S. et al. Implementation of OSI Presentation, ACSE and FTAM Protocol Software, and Its Evaluation, *Trans. IPS Japan*, **30**, 7 (1989) (in Japanese), 895–907.
12. NAKAKAWAJI, T. et al. Implementation and Evaluation of File Transfer Protocol Based on Standard Specification, *Trans. IPS Japan*, **29**, 11 (1988) (in Japanese), 1071–1078.
13. KATO, T. et al. Interconnection of Center Type Electronic Mail System ELMS and MHS, *Technical Report DSP 33-7, IPSJ* (May. 1987) (in Japanese).

(Received September 4, 1990; revised October 2, 1991)