Full-Automatic Implementation of Protocol Programs for OSI Application Protocols over ROSE

Toru Hasegawa, Akira Idoue, Toshihiko Kato and Kenji Suzuki
KDD R&D Laboratories
2-1-15 Ohara, Kamifukuoka-shi, Saitama 356-8502, Japan
(Phone) +81-492-78-7368
(Fax.) +81-492-78-7510
(E-mail) hasegawa@hsc.lab.kdd.co.jp

Key words: ASN.1 (Abstract Syntax Notation One), ROSE (Remote Operation Service Element), OSI, Automatic Program Generation

Abstract: Recently, the automatic program generation from protocol specification comes to be used in order to increase the efficiency of protocol program implementation. In the field of OSI, it can be applied successfully to the application protocol programs over ROSE. However, the conventional RO program generators have problems that they cannot generate complete protocol programs. This paper proposes a full-automatic implementation method of OSI application protocols over ROSE. Our RO program generator supports the program generation for more than one application protocols over ROSE such as MHS P2/P7, and enables the presentation context handling. As a result, we have succeeded to generate a complete MHS P2 / P7 protocol program. This paper describes the detailed design of our RO program generator and the results of implementation of P2/P7 program and its performance evaluation. The work which we did for the implementation was just to specify 370 line P2 / P7 protocol specification. The automatically generated P2 / P7 program can provide about 100 operations per second. Therefore, the proposed implementation method is considered to achieve as high performance as applicable to the practical usage.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35394-4_29

S. Budkowski et al. (eds.), *Formal Description Techniques and Protocol Specification, Testing and Verification*

1. INTRODUCTION

Recently, the automatic program generation from protocol specification comes to be used in order to increase the efficiency of protocol program implementation. The stub generator of RPC (remote procedure call) is a typical example (Birrel and Nelson, 1984). In the field of OSI, an ASN.1 (Abstract Syntax Notation One) (ITU Rec. 208 and X.209, 1987) compiler, which automatically generates encoder and decoder programs from the ASN.1 specification of Presentation and Application PDUs (Protocol Data Units), is an example of program generation (Hasegawa, Nomura and Kato, 1992) (Neufeld and Yang, 1990). Since the encoding rule of ASN.1 is quite complicated, the ASN.1 compilers are indispensable in the OSI protocol program implementation. However, the program generation by ASN.1 compilers is limited to encoder and decoder programs because ASN.1 specifies only the data type of PDUs. Implementers need to write the rest parts of programs.

The program generation can be applied successfully to the OSI application protocol programs working over ROSE (Remote Operation Service Element) (ITU Rec. X.219 and X.229, 1988), which is considered as an extension of RPC in the OSI environment. In the case of those protocols, such as MHS (Message Handling System) P2/P7 protocols (ITU Rec.X.413, X.419 and X.420, 1988) and CMIP (Common Management Information Protocol) (ITU Rec. X.710 and X.711, 1988), the protocol behaviors are defined in ROSE, and the data types of PDUs and service primitives provided for their users are defined in the *RO-notation*. Therefore, it is possible to generate the protocol programs from the specification in RO-notation, including the programs for PDU encoding and decoding, and for protocol behavior handling, similarly with RPC stub generators. For example, the ISODE software (Rose, 1990), which is a public domain OSI software widely used for an experimental purpose, uses a Remote Operation (RO) program generator called *rosy* (Remote Operations Stub-generator (YACC-based)) for protocols over ROSE (Rose, Onions and Robbins, 1991).

However, the conventional RO program generators have some problems that they cannot generate complete protocol programs, in the following points.

(1) In the OSI application layer, there are some cases that more than one protocols are used over ROSE. In MHS P2/P7 protocols, P7 protocol, which specifies the requests and responses between UA (User Agent) and MS (Message Store), transfers interpersonal messages defined by P2 protocol as user data. In CMIP, data types of managed object information transferred by CMIP PDUs are defined in different specification from CMIP itself. In such cases, the program generation needs to be performed for more

than one protocol specifications. But, the conventional RO program generators cannot handle these cases.
(2) In the OSI communications, the data types of PDUs are managed by the abstract syntax, and the presentation context containing the object identifier value of abstract syntax needs to be defined for an individual presentation connection. The conventional RO program generators do not generate protocol programs handling the presentation context.

We have taken account of those problems and implemented our RO program generator which can generate protocol programs fully supporting application protocols over ROSE. The above problems come from the fact that the RO-notation and ASN.1 specification cannot describe the mapping between PDUs specified independently nor assignments of the object identifier of abstract syntax. Therefore, we have extended the RO-notation and developed a program generator for the specification. Our RO program generator is also designed so as to generate protocol programs which have more flexible user programming interface than conventional RO program generators, and which realize high performance by avoiding data copying.

This paper describes the detailed design of our RO program generator which realizes the full-automatic implementation of OSI application protocols over ROSE, and the results that we applied it to the implementation of MHS P2/P7 protocol programs. Section 2 briefly introduces ROSE protocols and section 3 describes the design principles and overview of our RO program generator. Section 4 describes the detailed design and section 5 shows the results of implementing MHS P2/P7 protocol programs.

2. BRIEF INTRODUCTION TO ROSE

ROSE (ITU Rec. X.218 and X.219, 1988) is an application service element of OSI, and uses a protocol stack illustrated in Fig.1.. Among the protocols in Fig. 1, an application protocol and ROSE which are shaded are targets of automatic implementation. ROSE provides a request-response style communication for application protocols such as MHS P2/P7 and CMIP, and this request-response is called a remote operation. An entity of application protocol (a requester) requests an operation which has some arguments using ROSE. Receiving the operation, the peer entity (responder) sends back a response which has a result of the operation to the requester using ROSE.

The data types of PDUs of application protocols over ROSE are defined using four macros of RO-notation : *operation*, *bind*, *unbind* and *error*

macros. The operation macro is used to specify two kinds of application PDUs : an argument and a result of an operation. The bind and unbind macros are used to specify application PDUs transferred when an application association is established and released, respectively. The error macro is used to specify application PDUs which informs a requester of the error of operation. Among the application PDUs, those defined by the operation and error macros are transferred using ROSE protocol, and those defined by the bind and unbind macros are transferred using ACSE.

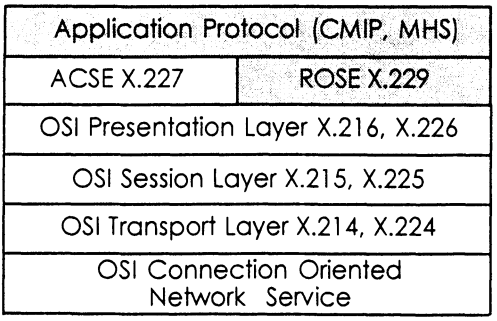

Figure 1. Protocol Stack

The ROSE protocol uses four ROSE PDUs : *ROIV (RO-INVOKE)*, *RORS (RO-RESULT)*, *ROER (RO-ERROR)* and *RORJ (RO-REJECT)*. A ROSE PDU transfers not only an application PDU as user data, but also the following ROSE protocol control information: *operation value* which specifies a type of operation, and an *invoke identifier* which is used to map the request and response of an operation.

It should be noted that the service primitives of the application protocols over ROSE have the same parameters as their PDUs, and therefore, the data types of the service primitives are also defined by RO-notation.

3. DESIGN PRINCIPLES AND OVERVIEW OF RO PROGRAM GENERATOR

3.1 DESIGN PRINCIPLES

We have adopted the following principles for our RO program generator.
(1) As described above, RO-notation is not powerful enough for describing a complete application protocol specification. Therefore, our RO program generator allows the following specifications to be added into RO-notation.

- A data type contained in *ANY DEFINED BY* type and *OCTET STRING* type. This specification defines the mappings between PDUs of more than one protocols over ROSE.

- The object identifier value for the abstract syntax for the PDU definitions.
- A role of operation, i.e. an operation is requested by the initiator of the application association, the acceptor, or both.

(2) Our RO program generator implements protocol programs supporting application protocols and ROSE. The protocol program interfaces between user programs for the upper layer side, and ACSE (Application Control Service Element) and PL (Presentation Layer) programs for the lower layer side. Since the application protocols over ROSE often require the invocation of an operation during waiting for a response of another operation, the subroutine call based user programming interface which is used in RPC stub and ISODE is not sufficient for flexible programming. Therefore, the user programming interface of the protocol programs automatically implemented adopts the exchange of individual service primitives through queues between the user programs and the protocol programs.

(3) As described above, the service primitives and PDUs of the application protocols contain the corresponding parameters, and the user programming interface of the automatically implemented protocol programs uses the service primitives. In order to avoid data copying, the encoder and decoder programs generated by our RO program generator encode PDUs from service primitive data and decode PDUs into service primitive buffer without using any intermediate working buffers.

3.2 OVERVIEW

(1) Our RO program generator consists of the extended ASN.1 compiler and the protocol behavior routines as shown in Fig. 2. Since individual application protocols use the same protocol behavior defined by ROSE, the protocol behavior is realized by pre-implemented library routines, i.e. the protocol behavior routines. On the contrary, the extended ASN.1 compiler generates the following application protocol specific programs and data from a specification in the RO-notation: primitive data types, ASN.1 encoder / decoder routines, and an operation table containing control data.

(2) The structure of generated program is shown in Fig. 3. The program provides the programming interface based on exchanging service primitives through interface queues to user programs. The data types of service primitives are generated by the extended ASN.1 compiler. The behavior routines perform the application protocol behaviors using the encoder / decoder routines and the operation table. The encoder / decoder routines are used to encode and decode ROSE PDUs and application PDUs. The

operation table holds the static information for all operations, a bind and an unbind. For each operation, the data includes an operation value, pointers to encoder and decoder routines for the PDU and error list.

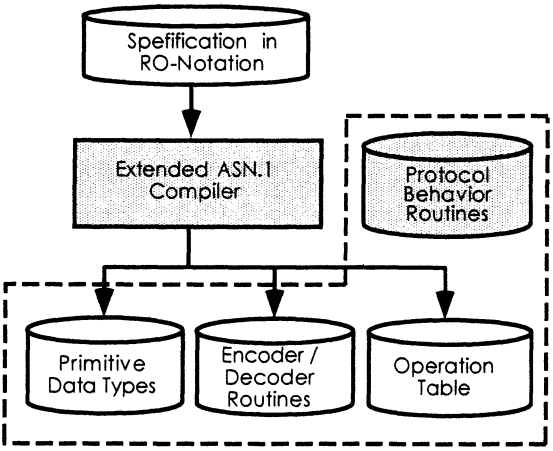

Figure 2. Structure of RO Program Generator

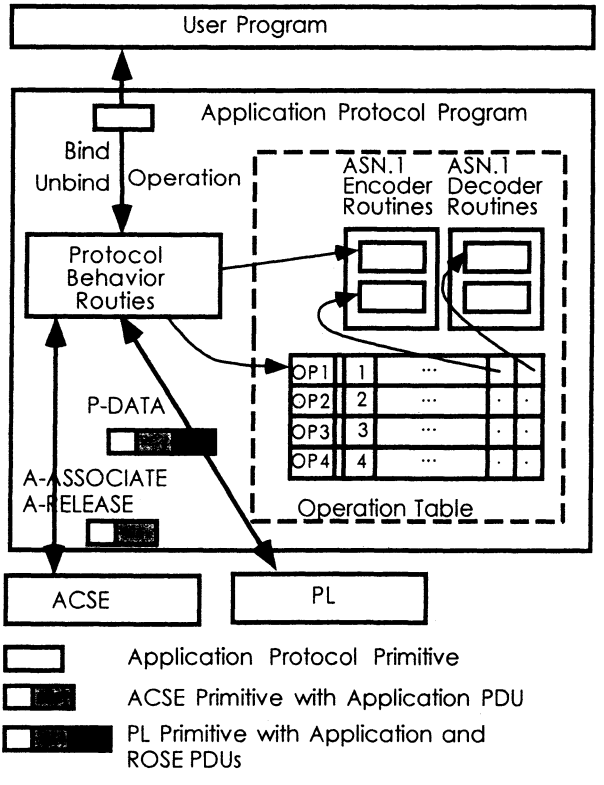

Figure 3. Structure of Generated Program

(3) The protocol behavior routines realize the application protocol processing in the following way.

- They use a control structure such that they receive a primitive from either a user program or a lower layer program (ACSE program or PL program), analyze the primitive, make a primitive and sends it to the lower layer program or the user program.
- When encoding a PDU from a service primitive and when decoding a PDU into a service primitive, they obtain a pointer to the encoder routine and the decoder routine by referring to the operation table.

They use two kinds of tables in order to manage the dynamic information. An association table is prepared for each application association. In each association table, a request-response table for mapping the request and response is provided.

4. DETAILED DESIGN

4.1 INPUT SPECIFICATION TO EXTENDED ASN.1 COMIPLER

An example of input specification to the extended ASN.1 compiler is shown in Fig. 4, which is a part of MHS P2 / P7 protocol specification. In this specification, a bind and an operation are defined by RO-notation extended as described above. *MSBind* is a bind, and *MessageSubmission* is an operation. The argument and result of MessageSubmission operation have *SubmissionArg* and *SubmissionRes* types, respectively, and the operation value is 3. These types are defined in ASN.1. For example, *SubmissionArg* type is defined using *SEQUENCE* type with two elements, *envelope* and *content*.

As described in section 3.2, the object identifier of the abstract syntax is specified at the line marked with *(a)* in Fig. 4. The line marked with *(b)* specifies the role of generated program for *MessageSubmission* operation. This operation is invoked by the initiator of an association. The line marked with *(c)* specifies that a P2 PDU whose data type is *InformationObject* is contained inside the *content* element whose data type is *OCTET STRING*. The line marked with *(d)* specifies the case that the *ANY DEFINED BY* type contains another data. In this case, the data types contained in the *values* element depends on the value of *atype* element whose data type is *OBJECT IDENTIFIER* type. The two lines followed by mark *(d)* specify pairs of the data type of contained data and the values of the *atype* element.

```
P2P7 DEFINITIONS
BEGIN
ABSSYN { 2 6 0 1 4 } -- (a) presentation context
MSBind ::= BIND -- bind
  .....
ABSSYN { 2 6 0 2 1 } -- (a)
MessageSubmission OPERATION
!! INITIATOR  -- requester -- (b)
  ARGUMENT  SubmissionArg
  RESULT    SubmissionRes
  ERRORS     { ..... }
::= 3  -- operation value
-- P7 PDU
SubmissionArg ::= SEQUENCE {
  envelope   Envelope,
  content    Content }
Content ::= OCTET STRING {type = InformationObject } --(c)
Attribute ::= SEQUENCE {
  atype   AttributeType,
  values ANY DEFINED atype -- (d)
    ( heading  SeqOfHeading { 2 6 1 7 0 },
      body     SeqOfBody { 2 6 1 8 0 },      ..... )
AttribueType ::= OBJECT IDENTIFIER
SeqofHeading  ::= SEQUENCE OF Heading
Heading  ::= SEQUENCE { ... }
-- P2 PDU
InformationObeject ::= CHOICE {
  ipm   IPM,    ..... }
-- ROSE PDU
ROSEApdus ::= .....
```

Figure 4. Example Specification (P2/P7)

4.2 SERVICE PRIMITIVE TYPE GENERATION

The extended ASN.1 compiler generates two service primitive data types in C language from each operation, bind and unbind definitions. One data type is for a request primitive and an indication primitive, and the other is for a response primitive and a confirm primitive.

A service primitive data type consists of the following elements:

- header : common information for all primitives such as a layer identifier, an indicator of a request, indication, response or confirm, and an association identifier.
- parameters of lower layer protocols : information needed to encode and decode ROSE PDUs and parameters exchanged between lower layers and a user program.
- application PDU

The data type of an application PDU is specified in ASN.1, and the C data type corresponding to that is generated by the extended ASN.1 compiler. As described above, the application PDU field is used both as a

working buffer in ASN.1 encoding / decoding and as a parameter of a primitive.

```
/* MessageSubmission Request / Indication Primitive */
struct MessageSubmission_req {
  prim_header_t   PrimHead;  /* header */
  long            InvokeId;  /* invoke identifier */
  SubmissionArg_t param;     /* application PDU :
                              argument of operation */ };
/* SubmissionArg Data Type */
struct SubmissionArg {
  Envelope_t   envelope;
  Contetnt_t   content; };
typedef struct SubmissionArg SubmissionArg_t;

/* MessageSubmission Response / Confirm Primitive */
struct MessageSubmission_rsp {
  prim_header_t   PrimHead;  /* header */
  long            InvokeId;  /* invoke identifier */
  long            resultFlag; /* Result : Success/Failure */
  union {
    SubmissionRes_t acc;     /* application PDU :
                                Response of operation */
    error_t         rej;     /* Error Response */;
                        } result; };
/* SubmissionRes Data Type */
struct SubmissionRes {
   …  };
typedef struct SubmissionRes SubmissionRes_t;
```

Figure 5. Service Primitive Data Types of Operation

Figure 5 shows the request and response primitive data types generated from an *MessageSubmission* operation specified in Fig. 4. The primitive data types are defined using C data type *struct* which holds the above mentioned information. For example, as for the request / indication primitive, *invokeID* element is an invoke identifier of ROSE PDU, and is used for encoding and decoding a ROSE PDU. *Param* element is a parameter of application PDU, i.e., an argument of operation, and the data type is specific to each operation. The response and confirm primitives include the following elements as well as invokeID element. *ResultFlag* element is a flag indicating whether the operation has been successfully performed by the responder or not. *Result* element is realized by C *union* structure and is either a result application PDU which is specific to each operation or an error application PDU. *Acc* element is the result application PDU, and *rej* element is the error application PDU.

It should be noted that the data types have the structure allocated in a contiguous memory area without using pointers as long as possible. The *SEQUENCE* and *SET* types which have nested structures do not use pointers to child components, but contain the values of child components themselves. In Fig. 5, C struct *SubmissioArg* corresponds to *SubmissionArg* of

SEQUENCE types, and have C struct themselves such as *Envelope_t* and *Content_t*.

The *SEQUENCE OF* and *SET OF* types which include ordered and unordered component values, respectively, have C data types which consist of an integer indicating a component value number, and a pointer to component values themselves. Figure 6 shows the C data types corresponding to *SeqOfHeading* type in Fig. 4. *Cnt* element and *data* element are the number of heading values and the pointer to the array of heading values. The memory area for storing heading values is allocated as a contiguous array, not as a list with individual heading values linked by pointers

```
struct Q_SeqOfHeading {
  u_long      cnt;    /* Number of Component Values */
  Heading_t   *data; /* Pointer to Array of Heading Values */ };
```

Figure 6. C Data Types of SEQUENCE OF Type

4.3 ENCODER / DECODER GENERATION

The extended ASN.1 compiler generates a C data type and encoder / decoder routines for a data type which is newly defined using structured types of ASN.1 such as *SEQUENCE* and *SET*. On the contrary, those for primitive types such as *INTEGER* and *OCTET STRING* types are pre-implemented as a library.

In order to deal with PDUs containing upper layer PDUs as user data, the mapping function is added to *ANY DEFINED BY* type and *OCTET STRING* type in the following way.

(1) A data type of upper layer PDU is defined using *DEFINED BY* phrase as shown in *(d)* of Fig. 4. The actual data type is dynamically chosen among the candidate data types. i.e, *SeqOfHeading* and *SeqOfBody* of Fig. 4. The data types are determined dynamically by an object identifier value at *atype* element. For example, when the object identifier value of type is {2 6 1 7 0}, the data type of *values* is *SeqOfHeading*.

(2) The data type of *ANY* with *DEFINED BY* phrase is a C data structure *union* which consists of all data types defined by *DEFINED BY* phrase, and it is generated by the extended ASN.1 compiler. Figure 7 shows the C data type generated from *Attribute* type which includes *ANY DEFINED BY* type. *Atype* element contains an object identifier value indicating data type of *values* element. *Values* element is a union of the C data types corresponding to the above candidate date types.

```
/* C data type for Attribute */
struct Q_Attribute
{   AttributeType_t atype;      /* Object Identifier */
    union                      /* Union of Candidate Types */
    {  SeqOfHeading_t heading;  /* Candidate Type 1 */
       SeqOfBody_t body;        /* Candidate Type 2 */
    }       values;       };
```

Figure 7. C Data Type for Attribute

(3) When encoding the parameter of ANY, an encoder routine first checks an object identifier value of *atype* element of working buffer, and then calls the encode routine corresponding to the object identifier value. On the contrary, when decoding, a decoder routine first decodes the *atype* element of PDU, and then calls the decoder routine corresponding to the decoded object identifier value.

```
void decodeAttribute(var, edata, index,
     error, ida)
  char   *var;     STRING  *edata;
  u_long   *index; ERROR    *error;
  ID      *ida;
{ Attribute  *q_var;
  long       cnt = 0;
  u_long     next_obj = 0;
  /* decoding identifier and length octets */
  cnt = skp_idlenN(edata, index, &next_obj, ida, error);
  next_obj += *index;
  .....
  q_var = (Attribute *) var;
  /* decoding type */
  decodeobjectN(&(*q_var).type, edata, index,
       error, id_infoN + 31, (u_long) 0); }
  if ((*error).errlevel < 2) {
    if (obj_chk (&(*q_var).type, &headingId) == TRUE) {
          decodeSeqOfHeading(
           &(*q_var).value.heading,
           edata, index, error, id_info + 44); }
    else if (obj_chk (&(*q_var).type, &bodyId) == TRUE) {
          decodeSeqOfBody(&(*q_var).value.body,
          edata, index, error, id_info + 45); }
    else {
       ERRSETN(error, FATALN, 10, ida, *index); }
  .....
}
/* Object Identifier Values : Constants */
OBJID headingId { { 2, 6, 1, 7, 0}, 5};
OBJID bodyId    { { 2, 6, 1, 8, 0}, 5};
```

Figure 8. Decoder Routine of Attribute Type

Figure 8 shows a part of decoder routine of *Attribute* type in Fig.4 which includes the above *ANY DEFINED BY* type. Octets to be decoded are stored in *edata* variable, and the decode result is set in *var* variable. First, the

decoder routine decodes identifier and length in the octets to be decoded using *skip_idlenN* library routine. Secondly, the decoder routine decodes object identifier of *atype* using *decodeobjectN* library routine. The decoded object identifier value is set in *atype* element of *var* variable. Thirdly, the routine compares the decoded object identifier value and candidate values using *obj_chk* library routine. The variables which hold candidate object identifier values are generated from an *ANY DEFINED BY* phrase by the extended ASN.1 compiler. *HeadingId* and *bodyId* variables in Fig. 8 are the examples, and they are used as arguments to *obj_chk* routine. Eventually, the decoder routine finds the decoder routine corresponding to the object identifier value, and calls the decoder routine, i.e., either *decodeSeqOfHeading* routine or *decodeSeqOfBody* routine which are generated by the extended ASN.1 compiler.

5. MHS P2/P7 IMPLEMENTATION AND COMMUNICATION EXPERIMENT

5.1 MHS P2/P7 PROTOCOL PROGRAM GENERATION

MHS P2/P7 protocol, which is defined over ROSE, is used to transfer an IPM (Interpersonal Message) between UAs (User Agents) and MSs (Message Stores). P2 protocol defines the format of IPM. P7 protocol defines the protocol between UAs and MSs, such as the submission of a message to an MS and fetch of a message stored in an MS.

We have developed the P2 /P7 program using the proposed implementation method. We have written a P2 / P7 specification in the extended RO- notation and generated the program using the extended ASN.1 compiler. The specification supports three operations : *list*, *fetch* and *deletion* of a message, and its size is 370 lines. From the specification, about 11 K line C programs are generated. The details are shown in Table 1. Since protocol behavior routines are about 2.4 K lines, the total P2 / P7 program size is about 13.4 K lines.

Among the 370 lines of the P2 / P7 specification, the 350 lines have been just copied from the specification in the ITU Recommendations (ITU Rec. X.413, X.419 and X.420), and just the 20 lines which specify the mapping of PDUs, presentation context identifiers and so on have been additionally written. It has taken just two days to write the specification and to generate the programs from the specification.

Table 1. Sizes of Specification and Generated Program

specification	lines	program	lines
RO-notation	80	operation table	120
		primitive data types	240
ASN.1	290	encoder / decoder	10,640

5.2 COMMUNICATION EXPERIMENT

In order to run the P2 / P7 program, UA and MS programs for testing have been developed. The P2 / P7 program, UA or MS program are built into the previously developed OSI program which supports from ACSE to LLC protocols. The total OSI program, whose structure is shown in Figure 9, is run as a single process on UNIX operating system. The execution of the above programs is controlled by a pseudo-kernel program which provides the scheduling function and inter-program communication function. For evaluating the generated program, we performed the following experiments.

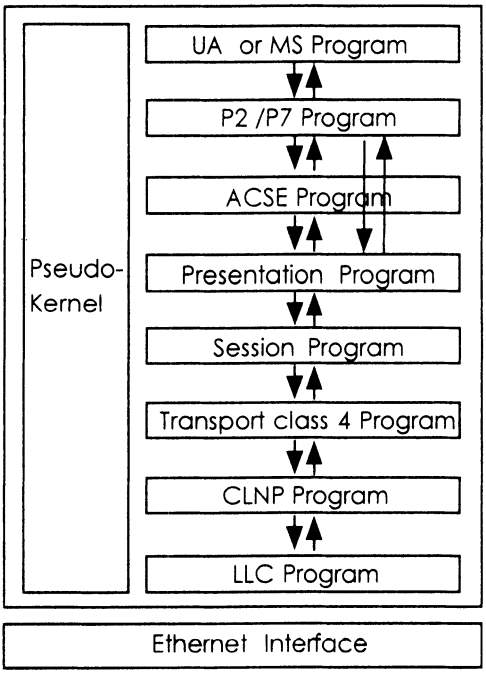

Figure 9. Program Structure

(1) Communication with Other MHS P2 / P7 Program

In order to validate the correctness of the automatically generated P2 / P7 program, we have performed the communication experiment between the

generated P2 / P7 program with MS program and a UA program which we previously manually developed. The MS program is run on a Sun SS-20 workstation and the UA program is run on a personal computer. These are connected via a serial line. The message submit, list and fetch operations are executed between the both programs, and the generated P2/P7 program with the MS program provides the correct MS behavior.

(2) Overall Performance Measurement

In order to measure the performance, the communication experiments are performed between the generated P2 / P7 program with UA program and the P2 /P7 program with MS program. The network configuration is shown in Fig. 10. The two Sun SS-20 workstations (SuparSPARC II 60 MHz) which are connected via Ethernet LAN are used.

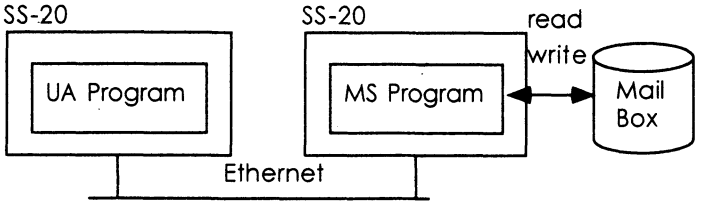

Figure 10. Network Configuration

We have measured the response times of list and fetch operations under the following conditions.

- list operation : The MS program searches the mail box, and returns a response which contains an originator name, a recipient name, a subject and a message identifier.
- fetch operation : The MS program reads a message whose body size is 1 K bytes, and returns a response which contains the message as a body.

The response times are measured by running 10,000 operations consecutively, and the measurement is performed ten times for each operation. A response time is a duration between the time when the UA program sends a request of operation and the time when it receives the response. The average response times of list and fetch operations are about 8.8 ms and 10.8 ms, respectively. As for a fetch operation, the processing tines of P2 / P7 programs of UA and MS sides are about 0.59 ms and 0.56 ms, respectively. As shown in the results, the automatically generated P2 / P7 program can support about 100 operations per second.

(3) Detailed Performance Measurement

We have evaluated the performance improvement caused by the proposed implementation method. The duration from the time of receiving a fetch request primitive from the MS program to the time of sending a P-DATA request primitive to the PL program is measured for both the proposed implementation method and the traditional method.

As for the proposed method, the duration is measured during the performance measurement of (2), and it is about 0.29 ms. On the contrary, as for the traditional method, we have developed a testing program for estimating the duration. The testing program uses a traditional method and a traditional ASN.1 compiler (Hasegawa, Nomura and Kato, 1992). The working buffer for an application PDU of fetch argument is not a parameter of a fetch request primitive. Therefore, the testing program converts the parameter of the fetch request primitive to the working buffer before it encodes using the ASN.1 encoder routine generated by the ASN.1 compiler. The duration is measured running the testing program on the same sun SS-20 workstation, and it is about 0.38 ms. The result shows that proposed direct encoding improves the P2/P7 program performance at about 30 %.

6. CONCLUSION

This paper has proposed a full-automatic implementation method of OSI application protocols over ROSE. We have realized a RO program generator consisting of the extended ASN.1 compiler and the protocol behavior routines. They support the program generation for more than one application protocols over ROSE such as MHS P2/P7, and enables the handling the presentation context. We have implemented MHS P2 / P7 protocol using our RO program generator. The results of implementation and communication experiment have made clear the following.

- The proposed method has achieved the full-automatic P2 / P7 protocol implementation. It took a few days to implement it since the required work was just to specify 370 line P2 /P7 protocol specification. Besides, since we did not need write any programming code at all, we did not encounter any program bug during the implementation.
- The automatically generated P2 / P7 program can provide about 100 operations per second. Besides, the processing time of P2 / P7 process is less than 1 ms. Therefore, the proposed implementation method is considered to achieve as high performance as applicable to the practical usage.

REFERENCES

Birrel, A. and Nelson, B. (1984) Implementing Remote Procedure Calls, ACM Trans. Comp. Syst. vol. 2, no. 1.

ITU Recommendations X.208 and X.209 (1987).

Hasegawa, T., Nomura, S. and Kato, T (1992) Implementation and Evaluation of ASN.1 Compiler, IPSJ Journal of INFORMATION PROCESSING, vol.15, no.2, 157-167.
Neufeld, G. and Yang, Y. (1990) An ASN.1 to Compiler, IEEE Trans. SE, vol.16, no.10, 1209-20..
ITU Recommendations X.219 and X.229 (1988).
ITU Recommendations X.413, X.419 and X.420 (1988).
ITU Recommendations X.710 and X.711 (1988).
Rose, M. T (1990) THE OPEN BOOK, Prentice-Hall.
Rose, M. T., Onions, J. P. and Robbins, C. J. (1991) The ISO Development Environment : User's Manual Volume 4 : The Application Cookbook.

BIOGRAPHY

Toru Hasegawa is a senior research engineer of High Speed Communications Lab. in KDD R&D Laboratories, Inc. Since joining KDD in 1984, he worked in the field of formal description technique (FDT) of communication protocols. From 1990 to 1991, he was a visiting researcher at Columbia University. His current interests include FDT, high-speed protocol and ATM. He received the B.S. and the M.S. Degree in information engineering from Kyoto University, Japan, in 1982 and 1984, respectively. He received IPSJ Convention Award in 1987.

Akira Idoue is a senior research engineer of High Speed Communications Lab. in KDD R&D Laboratories, Inc. Since joining KDD in 1986, he worked in the field of computer communication. His current research interests include implementation of high performance communication protocols and communication systems. He received the B.S. and M.E. Degrees of electrical engineering from Kobe University, Kobe, Japan, in 1984 and 1986 respectively.

Toshihiko Kato is the senior manager of High Speed Communications Lab. in KDD R&D Laboratories, Inc. Since joining KDD in1983, he has been working in the field of OSI, formal specification and conformance testing, distributed processing, ATM and high speed protocols. He received the B.S., M.E. and Dr. Eng. Degrees of electrical engineering from the University of Tokyo, in 1978, 1980 and 1983 respectively. From 1987 to 1988, he was a visiting scientist at Carnegie Melon University. Since 1993, he has been a Guest Associate Professor of Graduate School of Information Systems, in the University of Electro-Communications.

Kenji Suzuki is the executive vice president of KDD R&D Laboratories, Inc. Since joining KDD in 1976, he worked in the field of computer communication. He received the B.S., M.E. and Dr. Eng. Degrees of electrical engineering from Waseda University, Tokyo, Japan, in 1969, 1972 and 1976 respectively. He received Maejima Award from Communications Association of Japan in 1988, Achievement Award from the Institute of Electronics, Information and Communication Engineers in 1993, and Commendation by the Minister of State for Science and Technology (Persons of scientific and technological research merit) in 1995. Since 1993, he has been a Guest Professor of Graduate School of Information Systems, in the University of Electro-Communications.