Implementation support tools for OSI application layer protocols

Gregor v. Bochmann, Daniel Ouimet
Université de Montréal

and

Gerald Neufeld
University of British Columbia

Abstract

Formal specifications are a well-known technique for improving software development. In the context of OSI communication protocol standards, Formal Description Techniques (FDT's) have been developed for the description of communication protocols and services. In addition, a notation called ASN.1 is used for the descriptions of the data structures of protocol data units exchanges between communicating entities at the application layer. Existing FDT's, such as Estelle, LOTOS and SDL, do not include facilities to directly manipulate data structures defined in ASN.1. This makes using FDT's for distributed applications difficult. This paper deals with the integration of ASN.1 with Estelle, one of the FDT's, and the issues involved with the integration of corresponding implementation tools. It is shown how the encoding and decoding routines automatically generated from the ASN.1 definitions can be combined with implementation code semi-automatically generated from the Estelle specification of the protocol. Various choices for the implementation data structures are discussed. An application for a simple protocol is given.

1. Introduction

Standards for communication protocols and services are being developed for Open Systems Interconnection (OSI) [OSI 83] which are intended to allow the interworking of heterogeneous computer systems and applications. In this context, the protocol specifications are of particular importance, because they represent the standards which are the basis for the implementation and testing of compatible OSI systems [Boch 89d]. The standard specifications are usually written in natural language, augmented with formalisms such as state tables. In addition, formal description techniques (FDT's) have also been developed and used. The formal nature of these specifications make it possible to apply automated tools during the protocol development life cycle [Boch 87c].

The protocol specification defines the behavior of a protocol entity, as shown in Figure 1, in terms of its interactions with the local service user, called service primitives, and its interactions exchanged with the remote entity, called Protocol Data Units (PDU's). The following aspects of the behavior must be specified (for more details, see [Boch 89d]):

(a) temporal ordering of interactions,

(b) range of possible interaction parameters (data types of parameters),

(c) rules for interpreting and selecting values of interaction parameters for each particular instance of communication, and

(d) coding of the PDU's.

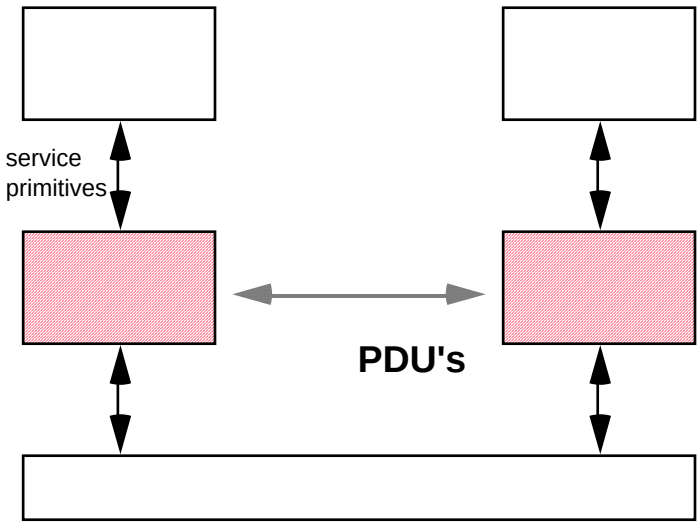

Figure 1- Communication Protocol Entity and Role of PDU's

In the case of OSI application layer protocols, the aspects (b) and (d) are usually described using a notation, called ASN.1 [ASN.1], which provides facilities for defining complex data types in terms of primitive data types and certain number of compositions, similar to data types in programming languages, such as Pascal. This notation also permits the definition of encoding rules [ASN.1 C] that are used to represent values of an application data type for transmission.

In order to reduce the effort required for the implementation of application layer protocols using this coding scheme, a number of ASN.1 support tools have been developed which provide for the automatic generation of encoding and decoding routines for PDU's defined in ASN.1. This covers, however, only the implementation issues related to the aspects (b) and (d) above. The other aspects of the implementation must be derived from the protocol specification by other means.

The FDT's can be used to describe the aspects (a), (b) and (c) above, but are less suitable for the coding aspect. A formal specification of a protocol given in one of the FDT's can therefore be used for the partial automation of the protocol implementation process [Boch 87h]. The text of the specification can be translated to obtain automatically part of the implementation code. In the case of the OSI protocols below the application layer, the coding aspect is defined in an ad hoc manner; reduction of implementation effort can only be obtained through code sharing for encoding routines. In the case of application layer protocols, however, because of the regular structure of the ASN.1 encoding scheme,

it is possible to automate the production of encoding routines. It is therefore advantageous to combine existing FDT implementation tools with the ASN.1 support tools. This is the topic of this paper.

Unfortunately, such a combination is not straightforward. The main difficulty comes from the fact that the FDT's have been developed separately from ASN.1 and also include language constructs for defining data types independent of ASN.1. It is therefore necessary to define a mapping between the ASN.1 data types and the corresponding data types of the FDT.

Section 2 of this paper provides a short overview of ASN.1. It also describes a translator which converts a given specification of the PDU's of an OSI application layer protocol into C data types as well as generating the encoding and decoding routines which translate between instances of these "C" language [Kern 88] data types and their corresponding bit-serial format used for transmission.

Section 3 gives a short overview of FDT's and related support tools. It also provides a discussion of the issues related to the integration of ASN.1 with the FDT's and identifies several strategies for doing so. One of these strategies involving the translation of the PDU structures from ASN.1 into the FDT is discussed in detail for the case of the FDT Estelle [Este 89] in Section 4.

Section 5 present a coordinated set of implementation support tools corresponding to the strategy described in Section 4. Some of the issues discussed relate to the use of an existing Estelle compiler and its incompatibilities with an evolving ASN.1 tool. Some experience with the use of this combined support tool is described in Section 6. This is followed with the conclusions in Section 7.

2. ASN.1 and its implementation tools

2.1. Overview of ASN.1

In a heterogeneous computing environment, computers differ in the way they represent values for primitive data types. For instance, one computer may represent characters in ASCII while another represents them in EBCDIC. It is also quite common for the representation of integers to vary in length from 8 to 64 bits depending on the computer architecture. In order to transfer data between such diverse computing environments, it is necessary to have a common understandable external data representation that is independent of any particular machine organization or programming language. A data value is translated (encoded) from its local representation into an external data representation, called the transfer syntax, before being sent on the communication link. It is then translated back (decoded) to a local representation at the receiving machine.

Although the requirement for a common representation of information exists at all layers in the OSI model, it is of particular importance at the application layer. The reason for this is that applications have a much greater need to transfer complex data structures.

Also, because there are many different applications, it is not possible to define a single representation as it is in the lower layers. Data structures are combined into a single unit called the Application Protocols Data Unit or APDU. An application may use many different APDUs. For example, the X.400 message handling system is an application for transferring mail message APDUs between computers. Each mail message APDU must be encoded into a standard external representation. Each mail message can have a very complex structure, including many fields such as the originator address and a set of recipient addresses. Each mail address itself is a complex structure.

In order to describe the structure of a APDU, an abstract notation is used, called "Abstract Syntax Notation One" or ASN.1 [ASN.1]. It is similar in many respects to the data typing facilities of modern programming languages such as Pascal, C or Modula II. However in the case of ASN.1 a richer set of data types exists. As such, there are two parts to an external data representation language: (1) an abstract syntax that defines the data types of the APDU's and (2) the transfer syntax that is the bit-serial representation of the instances or values of the data types.

ASN.1 as defined by ISO, is a formal abstract specification language which has been widely used in international standard specifications. Its encoding rules, the Basic Encoding Rules, defined also by ISO or translating ASN.1 data values into a transfer syntax has also been widely applied as an external data representation for transferring data over heterogeneous networks and computer systems. Several proprietary systems have also defined similar data specification languages. For instance, the language defined in Xerox's Courier protocol and the eXternal Data Representation (XDR) defined by Sun Microsystems, Inc. are also suchkind of data representation languages.

ASN.1's abstract syntax notation is defined in an analogous way to most modern programming languages. For example, Figure 2 is an example of how a personnel record could be defined.

```
PersonnelRecord ::=   [APPLICATION 0] SEQUENCE {
name                  OCTET STRING,
employeeNumber        INTEGER,
title                 OCTET STRING
dateOfHire            OCTET STRING, -- YYYYMMDD
children              SET OF ChildInfo }

ChildInfo ::=         SET {
name                  [0] OCTET STRING,
dateOfBirth           [1] OCTET STRING }
```

Figure 2

In this example, a personnel record is defined as a sequence of five types. A sequence in this context is similar to a record structure in Pascal. OCTET STRING is a variable length string of bytes. From the abstract syntax viewpoint there is no restriction on the length. In the case of field "dateOfHire", a comment is associated with the type

indicating the format. However, this is simply a comment for the application programmer. That is, ASN.1 does not enforce this restriction on the OCTET STRING. The data type SET OF indicates that there is a list containing an arbitrary number of children. The fact that it is a set means that the order of the elements has no significance. The typeChildInfo is also defined as a SET (but not SET OF). This means that the two fields - name and dateOfBirth - can be sent in either order. That is, name may be sent first followed by dateOfBirth or vice versa. In order for the application to know which field is sent, the fields must be tagged. In this case, a tag of 0 indicates the name, and a tag of 1 indicates dateOfBirth. Because the tag is unique (within the context of PeronnelRecord and ChildInfo) it can be used by the receiver to determine the order of the fields within ChildInfo.

There are many other facilities and options available in ASN.1. For example, it is possible to have SEQUENCE OF <type>, which indicates an arbitrary long array of <type>. The CHOICE facility, similar to a union type in C, allows the definition of alternate type. It is also possible to indicate that a field is optional. If a field is marked optional, then it may or may not be transmitted. In this case, the receiver must be able to detect that the field is missing.

Each data type in ASN.1 has a set of rules, called Basic EncodingRules (BER) [ASN.1 C], to translate it into its corresponding transfer syntax. Each data value to be transmitted is defined as either a primitive or a constructor. Generally if the data value contains other data values, such as PersonnelRecord in Figure 2, then it is defined as a constructor, otherwise it is a primitive. In either case, a value normally consists of a 3-tuple: an identifier, the length of the data, and the data itself. The first field identifies the type of data. For example, this field may indicate that the data is an INTEGER, or an OCTET STRING. The identifier also indicates that the data is a constructor or primitive. The length field can either contain the length directly or indicate that the length is unknown. In the latter case, an end-of-content flag is appended to the data field. If the data is a constructor, then the data would again contain a 3-tuple. With this technique, arbitrary complex data can be described and transmitted in an unambiguous manner.

2.2. Approaches to implementing ASN.1

A considerable amount of work has been done to develop approaches for the implementation of the BER and support tools for the ASN.1. Certain approaches only support the BER, by providing routines that translate between the BER-encoded form of PDU's used for transmission and the internal data structures. Other approaches provide in addition that received PDU be verified to satisfy the structure defined by the ASN.1 definition of the given protocol specification.

One can distinguish the interpretive and the compiling approaches. In the interpretive approach [Boch 86b, NBS ASN.1, Ohar 88, Naka 88], the ASN.1 definition is read by the ASN.1 tool during execution time and an internal representation, called the type tree, is constructed which is then used by a set of fixed, generic coding and decoding routines included in the software package. This approach is useful for systems that must adapt to a variety of different ASN.1 definitions, such as conformance testing tools. In the case of

the compiling approach [Yang 88, Hase 87, Hase 88, ISOD 89], the ASN.1 definition is read and processed during an ASN.1 compilation phase which reads the definition and produces programming language specific (i.e. C or Pascal) data type definitions for representing the internal value trees and a set of coding and decoding routines to be incorporated in the protocol processing program. It is important to note that the generated code is protocol specific and lends itself to more natural data structures for the internal representation of the PDU's (i.e. value trees). This approach is therefore particularly interesting for protocol implementation projects.

It is clear that the data structures adopted for the internal value trees have a strong impact on the ease with which the tool can be integrated with the rest of the protocol system. For example, the EAN implementation of the X.400 message handling system [Neuf 85] uses a generic tree structure, called ENODEs. An ENODE is a C structure that contains the identifier, length and pointer to the value of the data. Every data element is represented by an ENODE. If the data element is a constructor, then the pointer is to another ENODE(s). Using this method an APDU is represented internally by a tree of ENODES. Although this style of representing an APDU is very general, the syntactic structure of the data is lost to the application's programmer. For example, if in ASN.1 a count was described as: [APPLICATION 1] count INTEGER. This would be represented by an ENODE indicating the identifier(i.e., [APPLICATION 1] INTEGER), a length, and a pointer to the count, rather than "VAR count INTEGER" in Pascal or "int count" in C. The latter case is much more intuitive to the application programmer and is therefore the preferred way of representing APDUs.

In the case of the CASN.1 tool [Yang 88], which adopts the compilation approach, the compiler generates programmer-intuitive data types which can be used directly in the other parts of the protocol software. In order to encode an APDU, the programmer can simply call the encoding routine for the "top-level" data element of the value tree representing the APDU. This encoding routine calls other encoding routines for all the other data types in the APDU. When the top-level routine returns, the APDU is returned in the BER transfer syntax ready for transmission. Decoding an incoming APDU is very similar. On input, the top-level decoding routine is called, which calls all other descendant decoding routines. Each decoding routine is responsible for allocating the associated C data structure and storing the input value. On return,the top-level decoding routine points to the top-level C data structure.

The applications APDUs defined in ASN.1 are provided as input to the compiler which produces four different files; one containing the corresponding data structures(defs.c), two containing the encoding and decoding procedures (encode.c and decode.c), and one containing initialization code to set up the environment. Figure 3 shows the generated C data types for the PersonnelRecord defined in Figure 2.

```
typedef struct PersonnelRecord {
OCTS  *name;
int   employeeNumber;
OCTS  *title;
OCTS  *dataOfHire;
Set   Of(ChildInfo) children;
```

```
} PersonnelRecord;

typedef struct ChildInfo {
OCTS  *name
OCTS  *dateOfBirth;
} ChildInfo;
```

Figure 3

3. Formal description techniques and their relation to ASN.1

3.1. Overview of FDT's and existing support tools

Most protocol specifications are written in natural language, sometimes augmented with state tables and other forms of organized information. In addition, a number of formal techniques have been used to describe protocols in a more precise manner [Boch 89d]. The use of such formal specifications leads to the possibility of automating certain activities of the protocol development cycle [Boch 87c], such as the validation of the protocol specification [Pehr 89], the implementation process [Boch 87h], and the conformance testing of protocol implementations [Sari 89c].

In view of these advantages, ISO and CCITT have developed standardized specification languages, so-called formal description techniques (FDT's), which are intended to be used for the description of OSI protocols and services. They are respectively called Estelle [Este 89], LOTOS [Loto 89], and SDL [SDL 87] (for tutorial introductions, see [Budk 87, Bolo 87, Sara 87]). Although these languages were developed for use within OSI, they have potentially a much broader scope of application.

In Estelle, a specification module is modelled by an extended finite state machine (FSM). The extensions concerning the range of possible values for interaction parameters, and rules for interpreting and selecting values of these parameters (aspects (b) and (c) mentioned in the Introduction) are covered by type definitions, expressions and statements of the Pascal programming language. In addition, certain "Estelle statements" cover aspects related to the creation of the overall system structure consisting in general of a hierarchy of module instances. Communication between modules takes place through the interaction points of the modules which have been interconnected by the parent module. Communication is asynchronous, that is, an output message is stored in an input queue of the receiving module before it is processed.

SDL, which has the longest history, is also based on an extended FSM model. For the aspects (b) and (c) it uses the concept of abstract data types with the addition of a notation of program variables and data structures, similar to what is included in Estelle. However, the notation for the latter aspects is not related to Pascal, but to CHILL, the programming language recommended by CCITT. The communication is asynchronous and the destination process of an output message can be identified by various means, including process identifiers or channel names.

LOTOS is based on an algebraic calculus for communicating systems (CCS [Miln 80]) which includes the concepts of finite state machines plus parallel processes which communicate through a rendezvous mechanism which allows the specification of rendezvous between two or more processes. Asynchronous communication can be modelled by introducing queues explicitely as data types. The interactions are associated with gates which can be passed as parameters to other processes participating in the interactions. These gates play a role similar to the interaction points in Estelle. The aspects (b) and (c) are covered by an algebraic notation for abstract data types, called ACT ONE [Ehri 85], which is quite powerful, but would benefit from the introduction of certain abbreviated notations [Scol 87, Boch 89h] for the description of common data structures.

In contrast to the other FDT's, SDL was developed, right from the beginning, with an orientation towards a graphical representation. The language includes graphical elements for the FSM aspects of a process and the overall structure of a specification. The aspects (b) and (c) are only represented in the usual linear, program-like form. In addition, a completely program-like form is also defined called (SDL-PR) which is mainly used for the exchange of specifications between different SDL support systems.

In this paper, we are mainly concerned with issues related to the implementation process. The languages Estelle and SDL can be considered high-level implementation languages; in fact, a number of FDT compilers exist which translate formal specifications into program code written in a conventional programming language, such as Pascal, C or ADA. For Estelle, such systems are described in [Boch 87h, Este 87b, Vuon 88b].

3.2. Strategies for integrating ASN.1 and FDT's

In order to take advantage of the tools available for FDT's in conjunction with the ASN.1 notation used for OSI application layer protocols, together with the automatic generation of coding and decoding routines, it is necessary to clearly define a relationship between the ASN.1 data structure definitions and the corresponding concepts of the respective FDT. The relationship must be defined at two levels. First, the correspondence between the respective language constructs for defining data types (aspect (b) of the Introduction) must be defined. Secondly, issues related to the combination of the ASN.1 and FDT tools must be addressed.

3.2.1. Integration at the language level

The following three approaches can be envisioned for the integration of ASN.1 with an FDT: (1) substitution, (2) combination, and (3) translation.

In the substitution approach [Alve 87], the ASN.1 notation is used instead of the corresponding FDT concepts for describing the aspects (b) of the specification. It is important to note, however, that ASN.1 does not provide any notation for accessing individual elements contained in a given PDU data structure, and such a notation is required for writing those parts of the specification that describe the aspect (c). Therefore, this approach requires the definition of extensions to the ASN.1 notation.

Then it is possible to use such an extended notation instead of the data type definition and access constructs provided by the FDT.

In the case of the combination approach [Hase 88], the situation is similar, except that the constructs of the FDT remain valid and the extended ASN.1 notation is allowed as an alternative description method within the same language. Clearly, this leads to a duplication of functionality in the specification language.

In the translation approach [Boch 89h], no access extension for ASN.1 needs be defined. Instead, the ASN.1 notation is translated into equivalent constructs of the FDT. The formal specification of an OSI application layer protocol will therefore include the definition of the PDU structure as obtained by translation from the given ASN.1 definition contained in the protocol standard. For the access to the individual elements of this structure, the available constructs of the FDT can be used.

The substitution and combination approaches require a redesign of the FDT, which is a major undertaking. In the case of the translation approach, the FDT remains unchanged. In order to make this approach successful, however, it is necessary to define the translation in such a manner that the resulting PDU data type definitions are simple to read and natural for the designer of the FDT protocol specification. The details of the translation approach in the case of the FDT Estelle is further persued in the subsequent sections. For the case of LOTOS, the issues are discussed in [Boch 89h].

3.2.2. Integrating language tools in the case of ASN.1 translation

As mentioned above, various tools have been developed independently for ASN.1 and the BER, on the one hand, and for the FDT's on the other hand. The integration of these languages should therefore also lead to an integration of the associated tools. In the case of the translation approach described above, the main issue for tool integration is the compatibility of the respective implementation data structures used in the tools for representing the PDU's. As shown in Figure 4, the ASN.1 definition of the PDU, taken from the protocol standard document, is used for generating the ASN.1 encoding routines which translate between the BER transmission format and an internal data structure format used by the ASN.1 tool. The same ASN.1 definition is also translated into FDT data type definitions which become part of the formal protocol specification. This specification is then translated by the FDT tool. For the integration of these tools, it is therefore necessary that the implementation data structures representing the PDU's obtained by the translation from the FDT tool are the same as those used by the encoding routines generated by the ASN.1 tool.

Figure 4 - General Overview

An example of such an integration is discussed in the following sections. Another example, for the case of LOTOS, is described in [Boch 89h]. In the latter case, the existing systems were basically incompatible; the existing LOTOS interpreter used

internally Prolog data structures and LOTOS source code for externally representing constant data structures, while the ASN.1 tools used an internal value tree in C similar to the ENODE structure described in Section 2.2. Nevertheless, an integration was realized by providing the automatic generation of a translation module which transforms during run-time between the internal C value trees of the ASN.1 tool and the external source code supported by the LOTOS interpreter.

It is clear that these problems can be avoided when new tools are designed for an integrated ASN.1-FDT environment.

4. Translation from ASN.1 into Estelle/Pascal and C

4.1. Overview of translation difficulties

In this section we discuss in more detail the translation from ASN.1 into Estelle and C. Since ASN.1 only covers aspects related to the range of possible PDU parameter values and data structures of PDU parameters, the result of the translation will be data type definitions in the target language. In the case of Estelle, data types are defined in the same notation as in the Pascal programming language. The data typing facilities in many other modern programming languages are similar. This is also the case for C. In the following, we therefore mention the translation into C only in certain cases where its form is not evident from the corresponding discussion for Estelle.

Most constructs of ASN.1 are similar to constructs existing in programming languages, such as Pascal. For these constructs, a natural translation into Estelle and C is relatively straightforward, as discussed in Section 4.2. However, certain particular aspects are often difficult to match, for instance the arbitrary precision of integers in ASN.1 is not compatible with the fixed size integers in Pascal or C. Certain other constructs of ASN.1, such as SEQUENCE OF, have no equivalent in Estelle. It is therefore necessary to build data types in Estelle (and C) which correspond to these ASN.1 structures, as discussed in Section 4.3. In Section 4.4, the encountered translation difficulties are compared with the case that ASN.1 is translated into other specification languages.

It is important to note that Estelle is not an implementation language. For implementation, Estelle specifications can be automatically translated into parts of the implementation code [Boch 87h]. In Section 5, an integrated tool set for the implementation of ASN.1/Estelle specifications in the C language is described. In this case, the PDU definitions written in ASN.1 are translated by the ASN.1 tool directly into C. They are also translated into Estelle and incorporated into the complete Estelle specification of the protocol entity. This specification is then also translated by an Estelle compiler into the C language. As shown in Figure 4, it is important in this context, that the two PDU data type definitions in C obtained by the different translations be identical in order to make the generated encoding and decoding routines compatible with the C data structures used by the remaining part of the protocol entity. Issues related to this compatibility problem are discussed in Section 5.2 and 5.4.

4.2. ASN.1 constructs allowing a natural translation

4.2.1. Predefined data types

The predefined data types of INTEGER and BOOLEAN have a corresponding representation in Estelle. As mentioned above, the ASN.1 INTEGER has no upper bound. This is the same in Estelle, however, most Estelle compilers impose restrictions, corresponding to the target language, on the size of integers that can be handled, such as "integer" and "short" in C.

The ANY type of ASN.1 is a choice of an arbitrary number of types which is defined at runtime. This type can be represented in Estelle by the notation " ... " which means that the type is not yet defined and more information would be provided for an implementation. In the implementation in C, it would correspond to a pointer to any kind of value.

The ASN.1 types representing time (Generalized Time and Universal Time) can be represented in Estelle and C by a predefined record data structure, containing the fields year, month, day, hour, minute, second, time-difference, and zone, as shown in the Annex-1.

4.2.2. SEQUENCE and SET

An ASN.1 SEQUENCE or SET contains a certain number of elements of different types. The only difference between the two constructs is the order of transmission of these elements, which is sequential in the case of a SEQUENCE, and not defined in the case of a SET. This difference has only an impact on the encoding, not the meaning of the structures. Therefore they can be translated into identical structures in Estelle or C. The natural translation in Estelle is a "record", or a "struct" in C.

For example, the ASN.1 definition

```
Type1 ::=           SEQUENCE { a INTEGER,  b BOOLEAN }
```
would be translated into Estelle as:
```
Type1  =            record (*@SEQUENCE*)
                    a: INTEGER;
                    b: BOOLEAN; end;
```
or into C as:
```
typedef struct Type1 {
                    int a;
                    boolean b } Type1;
```
where "boolean" is a type already defined (see 5.4.1).

Note that the comment (*@SEQUENCE*) indicates that the translation was obtained from a SEQUENCE, and not a SET. This information is useful later for certain types of processing.

The translation of type structures from ASN.1 into Estelle also determines how the elements of the data types can be accessed by the body of the Estelle specification. Since Estelle includes a notation for accessing Estelle data structures, nothing has to be provided by the translation process. In the case of the above example, for instance, with an additional type definition

```
Type2 ::=                SET { x Type,  y BOOLEAN }
```

and a variable K2 of type Type2, one would use the Pascal "dot" notation to access the SEQUENCE x by the expression "K2.x", and the integer of x by "K2.x.a".

4.2.3. CHOICE

The ASN.1 type CHOICE indicates that a data item must be of one of several possible types. For instance, a value of the following type Type3 may be either an integer, a boolean or an octet string.

```
Type3 ::=       CHOICE {
        id1     INTEGER,
        id2     [UNIVERSAL 1] IMPLICIT BOOLEAN,
        id3     [APPLICATION 1] OCTET STRING }
```

The natural translation in Estelle (and Pascal) is a record with variants, such as the following:

```
Type3 = record (*@CHOICE *)
            case choice: integer of
            Type3_id1_tag:      (* = 0x2 *)
                    (id1:integer);
            Type3_id2_tag:      (* = 0x1 *)
                    (id2:(*@T [UNIVERSAL 1] IMPLICIT *) boolean);
            Type3_id3_tag:      (* = 0x60000001 *)
                    (id3:(*@T [APPLICATION 1] *) OCTSTRING);
            end;    (* of type Type3 *)
```

while in C the corresponding structure is a so-called union structure of the form

```
typedef struct Type3{           /* CHOICE */
        int         choice;     /* indicate the choice of data */
        union {     /* choices */
                    int         id1;    /* INTEGER */
                    bool        id2;    /* BOOLEAN */
                    OCTS        id3;    /* OCTET STRING */
                    }           data
        } Type3;
#define Type3_id1_tag 0x2
#define Type3_id2_tag 0x1
#define Type3_id3_tag 0x60000001
```

4.2.4. Tags

Tags are introduced into ASN.1 definitions in order to specify the identifier values that are inserted into the encoded form of the data values. They have no influence on the meaning of the data structures and their values can therefore be ignored in the translated data structures (possibly represented as comments in the Estelle or C translation). However, they are clearly important for the encoding and decoding routines, but also for the distinction between the different cases of a CHOICE data structure. The latter is the reason why identifiers have been introduced in the above translation representing the different possible tag values for the given ASN.1 CHOICE.[1] These identifiers can be used in the body of the Estelle specification (or the C-code implementation) to determine the value of a received CHOICE value.

4.3. Special translation structures

While the above translations are relatively straightforward, the following ASN.1 concepts have no immediate correspondence in Estelle, and must therefore be modelled by appropriate composed data structures.

4.3.1. Strings

ASN.1 distinguishes between OCTET STRING and BIT STRING. BIT STRING are packed to 8 bits per octet. There are several specializations of OCTET STRING, differing in the kinds of octets values that are allowed. For instance, an IA5String allows for the usual standard characters, NumericString only for numerical digits. These specializations can all be represented by the same manner in Estelle or C.

Estelle has no data type directly suitable for strings. Annex B1 of the Estelle standard [Este 89] proposes a set of procedures for manipulating values of a so-called "data_type", which is a kind of octet string. It would therefore be natural to consider the type"data_type" as the translation of the ASN.1 type OCTET STRING.

But in the case of translation into C, it is important to obtain an efficient implementation for string storage and manipulation. Whereas the "data_type" structure of Estelle keeps the entire string in an array, such an array structure is not very effective since first, the total length of a string is often not known in advance, and second, concatenation would require copying at least one of the two parts. Therefore we have adopted a data structure for strings which consists of one or more so-called "chunks", each containing a part of the string. An OCTET STRING in ASN.1 is therefore translated into a pointer to a structure (called OCTS) containing the following three fields: an octet string representing one chunk of the OCTET STRING, an integer specifying the length of that chunk, and a pointer to the next chunk. In order to simplify the translation from Estelle to C, we have adopted a similar data structure to represent OCTET STRING in Estelle. The

[1]This is not necessary if the target language has types as first-class values. Certain languages have statements for testing the type of a value.

corresponding data manipulation routines are listed in the Annex-1, and are similar to those defined for "data_type" in the Annex B1 of [Este 89].

In order to store and manipulate bit strings, we have adopted an approach similar to the case of octet strings. For Estelle and C, the ASN.1 type BIT STRING is translated into a pointer to a structure (called BITS) representing a chunk of the bit string and containing the following three fields: an octet string containing a chunk of the bit string (8 bits per octet), an integer equal to the number of bits in the chunk, and a pointer to the next chunk. The manipulation routines are similar to those for octet strings; their names have the prefix "b_" to distinguish them from the manipulation routines for octet strings, which have the prefix "o_".

4.3.2. SEQUENCE OF and SET OF

A value of type SEQUENCE OF <element type> is a sequence of values, each of type <element type>. A value of type SET OF <element type> is similar, except that the order of the elements has no significance. Because of this slight difference, the same representation could be used in Estelle or C.

There are several ways to represent such a sequence in Estelle. For instance, a list structure with pointers could be used as follows. Assuming the ASN.1 type

```
Type4 ::=      SEQUENCE OF Integer
```

the Estelle equivalent could be

```
Type4 =        ^ Type4_list;
Type4_list =   record
                    item:  integer;
                    next:  ^Type4_list
               end;
```

In order to simplify the manipulation of the list structure, we have adopted a different translation which is explained below. It is based on a generic list structure including additional information, and a set of standard list manipulation routines which can be used for all lists, independently of the type of the element type. A corresponding structure can be used in C.

The generic list structure in Estelle has the following form:

```
UNIV    = ...;          (* universal type, can represent any type *)
LIST_ITEM =             record
    item:               UNIV;
    next:               ^LIST_ITEM; (* next item in the list *)
    end;
LIST = record
    count:              integer; (* number of items in the list *)
    top:                ^LIST_ITEM;(* first item in the list *)
    current:            ^LIST_ITEM; (* next item to be accessed *)
    end;
LIST_OF  =              ^LIST; (* SET/SEQUENCE OF *)
```

The fields "count" and "current" are useful for the list manipulation. In order to keep them consistent, however, a set of manipulation routines have been defined (see Annex-1, primitives prefixed by C_List) which should be used in the body of the Estelle specification for accessing or updating the list structures.

Using the data structures above and the predefined manipulation routines, we may access a variable x of type SEQUENCE OF Integer (see Type4 above) as follows. The number of items in the list is given by "C_ListCount (X, result)" and the first element of the list is obtained by the expression "C_ListFirst (X, result_item)". However, the genericity of the data structure poses some problems in Estelle and Pascal because of strong typing. In fact, the items in the list structure are of type UNIV, which is "implementation dependent". They can be converted into elements of the appropriate type by using type casting routines generated by the ASN.1-Estelle compiler. For each type definition of the form SEQUENCE OF <element type>, the routines PutItem_<element type> and GetItem_<element type> are defined which provide type casting from <element type> to UNIV and UNIV to <element type> , respectively. For example, to obtain the integer which is the first element of the variable X considered above, we could write C_ListFirst (X, result_item); GetItem_Integer (result_integer, result_item). The use of these routines permits the programmer to write application code without knowing the internal implementation of these list structures.

A similar problem exists in C. The "item" field of the LIST_ITEM structure could directly contain the value of the element if the value fits into a single memory location, otherwise it would contain a pointer to the element. In order to hide this difference from the user, conversion routines as above can be used.

4.3.3. OPTIONAL

A element within a SEQUENCE or SET may be declared OPTIONAL. In this case, a SEQUENCE value may contain this element, or may not. The presence of the element within the data structure may be represented in different manners in Estelle or C. The following two approaches are possible:

(1) There is an additional boolean field included in the data structure for each element which may be optional. The boolean value indicates whether the element is present.

(2) If the element is represented by a pointer to the element value, the value NIL of the pointer means that the element is not present.

The first approach seems more natural. For example, the ASN.1 definition
Type5 ::= SEQUENCE { a INTEGER OPTIONAL; b BOOLEAN }
would be translated into:

```
Type5 =         record (*@SEQUENCE*)
    a_IsPresent:        BOOLEAN;
    a:          INTEGER;
    b:          BOOLEAN;
    end;
```

This approach has also been used for translation into LOTOS [Boch 89h].

4.4. Discussion

The above list of ASN.1 constructs covers the basic part of ASN.1. The so-called macros and certain recent extensions are not addressed, however. It is interesting to note that most features of ASN.1 can be translated in a natural manner into specification and programming languages, as shown above. The ASN.1 construct that encountered most difficulties is clearly the SEQUENCE/SET OF construct for which no corresponding concept exists in Estelle and most programming languages. This difficulty does not appear for the translation to LOTOS which contains a corresponding "String" construct [Boch 89h]. However, the data structure adopted for the translation into Estelle was partly influenced by considerations for implementation efficiency, since a natural translation into C implementation data structures was a concern. This concern is not addressed by the "String" data structure of LOTOS.

The implementation considerations for the C data structures lead to the use of pointers for various list structures. The access and update routines discussed in the subsections above have the property of completely hiding the pointer structures from the user. However, if the user wants to make a complete copy of a given PDU, which is represented in such a pointer data structure, he/she has to know its detailed structure. In order to avoid this difficulty, the ASN.1-Estelle compiler provides suitable copying routines for each ASN.1 type definition of the form SEQUENCE, SETand CHOICE. These routines have the form "s_copyXXX" were XXX is the name of the defined type (see also Annex-1). These routines make a copy of a variable of type XXX into another variable of that type by allocating the necessary memory for storing the data structures to be copied. For the list structures corresponding to SEQUENCE OF/SET OF , the predefined routines, C_ListCopy listed in Annex-1 can be used.

5. A coordinated set of support tools

5.1. Overview of the tool environment

In the following, we describe a set of support tools which have the purpose of providing an implementation environment for OSI Application layer protocols in conjuction with the use of the Estelle specification language. It is assumed that an ASN.1 description of the PDU's exist, and that an Estelle specification of the protocol is used in the implementation process. The support tools provide for the following steps:

(1) the automatic generation of encoding and decoding routines for the PDU's, and C type definitions.

(2) the automatic translation of the ASN.1 description of the PDU's into corresponding Estelle data structures, and the generation of declarations for the corresponding manipulation primitives, and

(3) the automatic generation of implementation code from the Estelle specification, and the generation of the algorithmic bodies for the corresponding manipulation primitives.

As indicated in Figure 4, this approach requires the compatibility between the data structures used in the encoding and decoding routines and the corresponding data structures obtained from the Estelle translation. A more detailed configuration of the support environment is shown in Figure 5. It is important to note that the Estelle data structures obtained by the translation step (2) above must be integrated into the Estelle specification of the protocol. As discussed in Section 4, these structures have an impact on the manner in which the processing of the PDU's can be described in the protocol specification. As indicated in Figure 5, the complete Estelle protocol specification is not obtained automatically. In order to simplify the implementation process, it would be useful if formal specifications of the OSI standard protocols would be available.

The ASN.1-Estelle implementation environment comprises three tools corresponding to the three steps identified above, which provide translation into the implementation language C. The first step is provided by the tool CASN.1 developed at the University of British Columbia [Yang 88]. The third step is provided by the NBS Estelle compiler [Este 87b]. The ASN.1 to Estelle translation of the second step was specially developed for this purpose, and is further described below. The system has been used for several implementations, as described in Section 6.

We believe that an integrated set of tools as described here will provide an implementation environment which reduces the number of programming errors in the implementation, because the PDU encoding and decoding is automatically generated, and the consistency with the PDU implementation data structures is automatically enforced.

5.2. Handling the incompatibilities between the ASN.1 and Estelle tools

The integration of two independently developed tools for steps (1) and (3) above into a single environment was the major challenge in the development of the ASN.1 to Estelle translation tool which must satisfy the compatibility requirement shown in Figure 4. While the Estelle compiler was fixed, the CASN.1 tool was still in development, and some requirements for compatibility of the C data structures could be taken into account. Among the different ASN.1 tools that we investigated, CASN.1 was the only one which showed sufficient compatibility with the C data structures generated by the Estelle compiler to make the integrated approach feasable.

A number of incompatibilities had to be bridged and a number of implementation restrictions imposed by the two existing tools had to be accomodated, as described below. The necessary adjustments were made for each of these cases by using one or several of the following approaches:

(a) Adjusting the ASN.1 to Estelle translation (step (2)) accordingly. The objective of realizing a "natural" translation between the two languages limits the range of possible changes.

(b) Adjusting the C data structures used by the CASN.1 tool. The possibility of introducing an option for generating structures compatible with the ASN.1-to-Estelle translation and the NBS Estelle compiler was considered a possibility, but finally not used.

Figure 5 goes here; it comes from a PowerPoint document

(c) The automatic introduction of changes into the data structures generated by the CASN.1 tool. This approach was adopted for certain incompatibilities, and was realized through the automatic generation of text editing commands during step (2) of Section 5.1, and the automatic execution of these commands on the source code generated in step (1). The "SED" (standard Unix stream editor) is used for this purpose.

5.3. The ASN.1 to Estelle translation

This translation corresponding to step (2) above, is performed by the ASN1ESTL compiler which executes three compilation passes. The first pass is done for lexical and syntaxic analysis and some semantic checking. It also makes a list of all types with their interdependencies, as needed by the next pass. A textual alignment of the ASN.1 definition is also performed which is useful for pretty printing. The second pass reorders all types such that a type is not used before its definition. Recursion definitions in ASN.1 are not supported by the compiler. The third pass does additional semantic checking and translates the ASN.1 code into Estelle. It also generates some constant and header declarations in C to be combined later with the C code generated by the other tools (see Figure 5). In addition, it generates the necessary Estelle primitives and C routines for manipulating the PDU data structures, as described in Section 4.4, and includes a file of

predefined type declarations. Finally, it also generates some text editor commands for modifying the source code obtained from the CASN.1 tool in order to obtain compatible data structures.

The first and third passes of the compiler were made using YACC and LEXcompiler generation tools. The lexical and syntactic definitions amount to about 600 lines of text. The translation and semantic checking functions are written in C. They amount to appoximately 100 and 3000 lines, for the two passes respectively. The second pass is a simple C program of about 500 lines.

5.4. Particular compatibility problems

Among the various smaller and larger problems related to the compatibility of the data structures generated by the CASN.1 tool and the NBS Estelle compiler, we mention in the following a certain number which seem to be the most interesting ones.

5.4.1. BOOLEAN translations

Because no translation exists in C for a boolean, each tool has to choose a C representation for this type. The Estelle compiler has chosen an integer, and CASN.1 a short integer. A short integer is 16 bits while an integer is 16 or 32 bits (machine dependant). We have redefined the type "boolean" as "short int" in the generated C code.

5.4.2 CHOICE and OPTIONAL: translation incompatibility

CASN.1 translates a CHOICE into C as a union structure, and the ASN1ESTL translation into Estelle is a variant record. It would seem natural that the variant record would be translated into the same C union structure, however, the given Estelle compiler [Este 87b] translates it into a plain record, without a "union". This incompatibility is solved by modifying the C code generated by the CASN.1 tool, as explained in point (c) of Section 5.2. Certain references to these data structures in the coding and decoding routines are also changed.

A similar modification is made to change the translation of OPTIONAL, since the CASN.1 tool uses the approach (2) of Section 4.3.3 while our ASN.1 to Estelle translation uses approach (1).

5.4.3 "current" interference.

As explained in Section 4.3.2, the "current" field in a LIST data structure can be used by the specification to access items of the LIST in an orderly manner. It is noted that a similar field is used for the internal processing of the encoding routines. In order to avoid interference between these two functions, two separate "current" fields are introduced in these list structures.

6. Some practical experience

Some simple applications of the tools described above were made during their development. We describe in the following the application of a version of the alternating bit protocol (ABP). The sender entity of this protocol sends data PDU's to the receiver entity and receives acknowledgement PDU's from the latter. Each data PDU contains, in addition to the user data, a sequence number which has the alternating value of 0 and 1. The acknowledge PDU's contain only a sequence number. The acknowledgement PDU with value 0 acknowledges the correct reception of a data PDU with sequence number 0, or asks for retransmission if the last data PDU sent had the sequence number 1. A complete Estelle specification of the ABP is given in Annex B2 of [Este 89]. We adapted this specification by introducing the following PDU definitions written in ASN.1.

```
AB-PDU DEFINITIONS ::=  BEGIN
Npdu-type ::=          CHOICE
{   data-pdu           [APPLICATION 1] Data-pdu-type,
ack-pdu                [APPLICATION 2] Ack-pdu-type }

Data-pdu-type ::=      SEQUENCE {
ndata                  U-Data-type,
seq                    Seq-type }

Ack-pdu-type ::=       SEQUENCE {
seq                    Seq-type }

U-Data-type ::=        SET OF IA5String

Seq-type ::=           INTEGER { seq0 (0), seq1 (1) }
END --                 of AB_PDU definitions --
```

Another change in the ABP specification was made, namely the introduction of an additional Estelle module as shown in Figure 6. The Alternating_bit_type module of Figure 6 corresponds to the description of the ABP as given in [Este 89], while the sole purpose of the Mapping module is the encoding and decoding of PDU's. The reason for introducing a separate module for this purpose is that the implementation is simplified if the received PDU's are decoded before their parameters are accessed by the Alternating_bit_type module. In fact, the Estelle PROVIDED clauses of certain transitions in this module access the parameters of received PDU's in order to determine whether the transition in question can be executed or not. It is assumed that at this point the parameters of the received PDU are accessible. It is noted that a similar Mapping module is also used in the specification of the Transport protocol [Boch 90] where it has the additional purpose of handling the multiplexing of several Transport connections over a single Network connection.

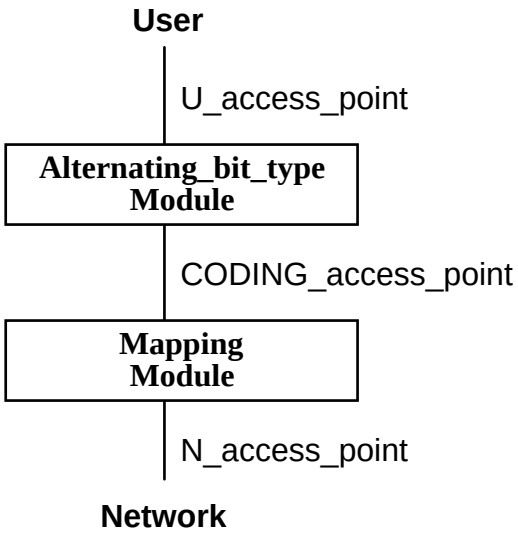

Figure 6 - Structure of Alternating Bit Specification

The complete Estelle specification of the ABP used for our experiment is given in Annex-2. The declaration part of the specification includes two source files generated by the ASN1ESTL compiler (step(2) of Section 5.1). The file "asn1estl/const.e" shown in Annex-1 contains the constants for tags, named integers and named bits in the ASN.1 definitions. The file "asn1estl/originalfilename.e" shown in Annex-1 contains corresponding type and procedure definitions.

The latter file consists of five major parts, respectively in this order: (1) the predefined types for the CASN.1 tool, such as OCTS, BITS, LIST, etc., (2) the predefined types for ASN.1 such as IA5String, GeneralString, UTCTime, etc., (3) the translated types from the PDU definition, (4) the predefined primitives for manipulating predefined structures, and (5) useful primitives corresponding to the PDU definitions, such as coding and decoding primitives, mapping primitives for List items, primitives for duplicating or releasing structures like SEQUENCE, SET, CHOICE, etc.

7. Conclusions

The ASN.1 notation is used for the description of the data structures of protocol data units (PDU's) for most OSI Application layer and related protocols. However, this notation lacks facilities for defining the access to particular components of the data structure and any actions associated with the operations to be performed within the protocol entity. The latter aspects of the protocol specification are usually defined in natural language.

In order to take full advantage of the automated tools developed for the validation, implementation and testing of communication protocols [Boch 87c], it is necessary to use formal or semi-formal specifications of the protocols. The question of combining ASN.1 with existing formal description techniques (FDT's) therefore arrises. The approach of translating ASN.1 into Estelle data types and the use of full Estelle protocol

specifications for the partial automation of the implementation process is discussed in this paper.

This approach has the following characteristics:

(a) Given an Estelle specification of the protocol, including the PDU descriptions obtained automatically by the translation of the ASN.1 definitions in the OSI standard, the implementation process is largely automated. First, the encoding and decoding routines for the PDU's, based on the ASN.1 encoding scheme, are automatically generated. Second, the major part of the code representing the processing, by the protocol entity, of the PDU's and service primitives is automatically obtained based on the other parts of the Estelle specification.

(b) This automation provides a faster and more reliable implementation process. The direct use of the standard ASN.1 definitions, and possibly of an Estelle specification which has been validated in respect to the protocol standard, implies that the resulting implementation is closely related to the protocol standard and it can be expected that few problems will arise during the conformance testing of the protocol implementation.

(c) It is possible to take advantage of already existing implementation tools for ASN.1 and Estelle, and combine them by developing a translation tool from ASN.1 to Estelle, and making minor adjustments to the existing tools. The resulting tool set provides an implementation environment for OSI Application layer protocols.

(d) The Estelle protocol specification used in this context uses the PDU data structures obtained by the automatic translation from the ASN.1 protocol definition. The development of such a specification is outside the scope of this paper. It is conceivable that such a specification would be given as an annex of the protocol standard. If a validated formal protocol specification is available, such an implementation approach is clearly very promising.

(e) It is also possible to use ASN.1 type definition for other purpose than PDU definition, such as service parameters or internal types in a specification. This has the advantage that the list manipulation routines provided by the ASN.1 support can also be used for processing these data structures; thus simplifying the implementation. In certain application, such as the OSI Directory, ASN.1 structured data may also be stored in a database.

As the discussions in this paper show, the translation from ASN.1 to Estelle is straightforward for most aspects of the notation, but certain difficulties arise in relation with the following points:

(a) Certain aspects of the ASN.1 notation, such as tags and the distinction between SETs and SEQUENCEs, have no semantic meaning, but are only relevant for the coding aspects of the PDU's.

(b) Estelle (and Pascal on which the type system of Estelle is based) has no facilities for generic data types. This limitation is an impact on how the list structure of the ASN.1

construct SEQUENCE/SET OF can be translated into Estelle. The approach described in Section 4.3.2 seems a reasonable solution to this problem, however, one sees that implementation-oriented features must be considered in the Estelle data types in order to obtain a reasonable automatic translation into implementation code.

(c) Similar problems arise in relation with the handling of strings and optional fields.

Our preliminary experience with the here described semi-automatic implementation approach seems to indicate that it is very promising to integrate the ASN.1 notation with other formal languages that can be used to formally describe the other aspects of the protocol definitions. We are presently working on the application of this implementation approach to the OSI Association Control (ACSE) protocol. It would be interesting to provide extensions to the tools for handling the ROSE ("Remote Operations", ISO IS9072) macros used by many OSI application layer protocols.

While this paper limits its attention to the use of Estelle as an FDT to be combined with ASN.1, a similar approach can be persued for the translation of ASN.1 into other languages, such as LOTOS and SDL. The semi-automatic implementation approach described in this paper could also be adopted with SDL, instead of Estelle. In fact, SDL has certain similarities with Estelle and allows for semi-automatic implementation of most of its language features. In particular, it includes type constructs similar those of Pascal. Therefore the translation of ASN.1 described in Section 4 could be adapted for SDL as the target language.

Acknowledgements

This work would not have been possible without the dedication and competence of Yueli Yang who designed and implemented the CASN.1 tool. Financial support from the National Science and Engineering Research Council of Canada is acknowledged.

References

[ASN.1 C] ISO IS 8825, "Information Processing - Open systems Interconnection - Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).

[ASN.1] ISO IS 8824, "Information Processing - Open systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1).

[Alve 87] "Formal Methods Applied to Protocols (FORMAT): A Framework for the Underlying Concepts of Communications Protocols", Report no.4 to the Alvey Directorate, UK, February 1987.

[Boch 86b] G.v.Bochmann, M.Deslauriers and S.Bessette, "Application layer protocol testing and ASN1 support tools", Proc. IEEE GLOBECOM Conf., Houston, Dec. 1986, pp. 767-771.

[Boch 87c] G.v.Bochmann, "Usage of protocol development tools: the results of a survey" (invited paper), 7-th IFIP Symposium on Protocol Specification, Testing and Verification, Zurich, May 1987.

[Boch 87h] G.v.Bochmann, G.Gerber, and J.M.Serre, "Semiautomatic implementation of communication protocols", IEEE Tr. on SE, Vol. SE-13, No. 9, September 1987, pp. 989-1000 (reprinted in "Automatic Implementation and Conformance Testing of OSI Protocols", IEEE, edited by D.P.Sidhu, 1989).

[Boch 89d] G.v.Bochmann, "Protocol specification for OSI", to be published in Computer Networks and ISDN Systems.

[Boch 89h] G.v.Bochmann and M.Deslauriers, "Combining ASN1 support with the LOTOS language", Proc. IFIP Symp. on Protocol Specification, Testing and Verification XI, June 1989, North Holland Publ.

[Boch 90] G.v.Bochmann, "Specifications of a simplified Transport protocol using different formal description techniques", to be published in Computer Networks and ISDN Systems.

[Bolo 87] T.Bolognesi and E.Brinksma, "Introduction to the ISO Specification Language Lotos", Computer Networks and ISDN Systems, vol. 14, no. 1, pp.3- , 1987.

[Budk 87] S.Budkowski and P.Dembinski, "An introduction to Estelle: a specification language for distributed systems", Computer Networks and ISDN Systems, vol. 14, no. 1, pp.25- , 1987.

[Ehri 85] H.Ehrig and B.Mahr, "Fundamentals of Algebraic Specifications 1", Springer Verlag, 1985.

[Este 87b] NBS, "User guide for the NBS prototype compiler for Estelle", Final Report, Report no. ICST/SNA - 87/3, Octobre 1987.

[Este 89] ISO IS9074 (1989) "Estelle: A formal description technique based on an extended state transition model".

[Hase 87] T.Hasegawa, et al., "Translator from protocol specification based on Estelle and ASN.1 to Ada", Technical Report on Information Network of IECE of Japan, IN87-74, 1987.

[Hase 88] T.Hasegawa, et al., "Automatic ADA program generation from protocol specifications based on Estelle and ASN.1", Proceedings of International Conference on Computer Communications (ICCC), J.Raviv editor, Haifa, Israel, 1988.

[ISOD 89] "ISODE-5.0, ISO Development Environment", Software Package by The Wollongong Group, Palo Alto CA, USA, 1989.

[Kern 88] B. W. Kernighan and D. M. Ritchie, "The C Programming Language", second edition, Prentice Hall Software Series, New Jersey, 1988.

[Miln 80] R.Milner, "A calculus of communicating systems", Lecture Notes in CS, No. 92, Springer Verlag, 1980.

[NBS ASN1] ASN1 support tools developed at NBS, complete with user's manual, Gaithersburg, US.

[Naka 88] T.Nakakawaji, K.Katsuyama, N.Miyauchi and T.Mizuno, "Development and Evaluation of APRICOT (Tools for Abstract Syntax Notation One)", Proceedings of the Second International Symposium on Interoperable Information Systems ISIIS '88, Tokyo, 1988, pp.55-62.

[Neuf 85] G.W.Neufeld, J.Demco, B.Hilpert, R.Sample, "EAN: An X.400 message System", Second International Symposium on Computer Message Systems, North Holland ,September 1985, pp.1-13.

[OSI 83] Special issue on Open Systems Interworking, Proc. of the IEEE, Dec. 1983.

[Ohar 88] Y.Ohara, T.Suganuma and S.Senda, "ASN.1 Tools for Semi-automatic Implementation of OSI Application Layer Prorotols", Proceedings of the Second International Symposium on Interoperable Information Systems ISIIS '88, Tokyo, 1988, pp.63-70.

[Pehr 89] B.Pehrson, "Protocol verification", to be published in Computer Networks and ISDN systems.

[SDL 87] CCITT SG XI, Recommendation Z.100 (1987)

[Sara 87] R.Sarraco and P.A.J.Tilanus, "CCITT SDL: Overview of the language and its applications", Computer Network and ISDN Systems, 13 (1987), pp. 65-74.

[Sari 89c] B.Sarikaya, "Conformance Testing: Architectures and Test Sequences", Computer Networks and ISDN Systems 17 (1989), pp. 111-126.

[Scol 87] ISO 97/21 N1540, "Potential enhancements to Lotos", 1986.

[Vuon 88b] S.T.Vuong et al., "An Estelle - C compiler for automatic protocol implementation", Proc. IFIP Symposium on Protocol Specification, Testing and Verification, Atl. City, 1988.

[Yang 88] Y.Yang, "ASN.1-C Compiler for Automatic Protocol Implementation", Master Degree Thesis, Department of Computer Science, University of British Columbia, October 1988, 111pp.

```
=====================================================================
Annex-1: Generated files from ASN1-Estelle tool.

This is the generated file from ASN1-Estelle compiler for
Alternating-Bit ASN.1 PDU definition given in Section 6.

-----FILE const.e
       (* Constants defined by ASN.1->Estl compiler *)
const
       (* some predefined constants *)
(*  ..... some constants deleted ..... *)
       (* list of named integers for Seq_type: *)
seq0   = 0;
seq1   = 1;

       (* tags constants for CHOICE type Npdu_type *)
Npdu_type_data_pdu_tag = ANY INTEGER; (* will be defined in c_include.h as 0x60000001 *)
Npdu_type_ack_pdu_tag  = ANY INTEGER; (* will be defined in c_include.h as 0x60000002 *)

-----FILE ab-pdu.asn1.e
type
              (* predefined types *)
   anything = ...;     (* type used for ANY *)
   Along = ...;
   cstring = ...;      (* string of character *)
   bstring = ...;      (* string of byte *)
   uint_32 = ...;
   UNIV = ...;
   idnull = INTEGER;  (* null type is represented by integer *)
   ASN1_DATA_TYPE = record
       (* type used to represent a value encoded with BER *)
                 octetseq:   bstring;
                 length: integer;
               end;
   PTR_BITS = ...;     (* will be ^BITS *)
   BITS = record       (* BIT STRING *)
             next: PTR_BITS;
             len: Along;
             data: cstring;
          end;
   BITSTRING = ^BITS;
   PTR_OCTS = ...;     (* will be ^OCTS *)
   OCTS = record       (* OCTET STRING *)
             next: PTR_OCTS;
             len: Along;
             data: cstring;
          end;
   OCTSTRING = ^OCTS;
   PTR_LIST_ITEM = ...;       (* will be ^LIST_ITEM *)
   LIST_ITEM = record
             dummy: uint_32;
             item: UNIV;
             next: PTR_LIST_ITEM;
             end;
   LIST = record
             count: uint_32;
             top: ^LIST_ITEM;
             casn1_nextcurrent: ^LIST_ITEM;
             end;
   LIST_OF  = ^LIST;  (* SET/SEQUENCE OF *)
   ASN1TIME = record
             year:   integer; (* year  :  19** or **, (0 .. ~) *)
             month:  integer; (* month :  1 .. 12 *)
             day:    integer; (* hour  :  1 .. 31 *)
             hour:   integer; (* day   :  0 .. 23 *)
             minute: integer; (* minute:  0 .. 59 *)
             second: real;    (* second:  0 .. 59 *)
             diff:   real;    (* difference between local time and UTCtime *)
             zone:   integer; (* flag for time form: (0,1,2,3,4) *)
                  (* UTC_Z_TIME for UTC time with Z
                     UTC_D_TIME for UTC time with differential
                     GNL_Z_TIME for Generalized time with Z
                     GNL_D_TIME for Generalized time with differential
                     GNL_L_TIME for Generalized local time *)
             end;
```

```
   OCTET =     0..255;
   BIT =       0..1;
   LEN_TYPE =  0..MAXDATA;
   ID_TYPE =   1..MAXDATA;
   DATA_TYPE =record
                L: LEN_TYPE;
                D: array[ID_TYPE] of OCTET;
        (* each element of D can be used to store an OCTET, a BIT or a CHAR.
           for CHAR, CHR() and ORD() functions should be used *)
                end;

NumericString (*@T [UNIVERSAL 18] IMPLICIT *)       = OCTSTRING;
PrintableString (*@T [UNIVERSAL 19] IMPLICIT *)     = OCTSTRING;
TeletexString (*@T [UNIVERSAL 20] IMPLICIT *)       = OCTSTRING;
T61String (*@T [UNIVERSAL 20] IMPLICIT *)    = OCTSTRING;
VideotexString (*@T [UNIVERSAL 21] IMPLICIT *)      = OCTSTRING;
IA5String (*@T [UNIVERSAL 22] IMPLICIT *)    = OCTSTRING;
GraphicString (*@T [UNIVERSAL 25] IMPLICIT *)       = OCTSTRING;
VisibleString (*@T [UNIVERSAL 26] IMPLICIT *)       = OCTSTRING;
ISO646String (*@T [UNIVERSAL 26] IMPLICIT *)        = OCTSTRING;
GeneralString (*@T [UNIVERSAL 27] IMPLICIT *)       = OCTSTRING;
UTCTime (*@T [UNIVERSAL 23] IMPLICIT *)      = ASN1TIME;
GeneralizedTime (*@T [UNIVERSAL 24] IMPLICIT *)     = ASN1TIME;

                        (* Start of the translation *)
U_Data_type    = (*@SetOf *) LIST_OF; (* of IA5String *)
       (*@ONS non-significative order *)
       (* primitives to map/unmap Items of this type will be declared as:
          GetItem_IA5String and PutItem_IA5String procedures
          at the end of types *)

Seq_type       = integer;
       (* list of named integers for Seq_type:
       seq0    : 0
       seq1    : 1 *)

Data_pdu_type  = record (*@SEQUENCE *)
                        (*@OS significative order*)
       ndata:  U_Data_type;
       seq:    Seq_type;
       end; (* of type Data_pdu_type *)

Ack_pdu_type   = record (*@SEQUENCE *)
                        (*@OS significative order*)
       seq:    Seq_type;
       end; (* of type Ack_pdu_type *)

Npdu_type      = record (*@CHOICE *)
       case choice: integer of
       Npdu_type_data_pdu_tag:       (* = 0x60000001 *)
         (data_pdu:   (*@T [APPLICATION 1] *) ^Data_pdu_type);
       Npdu_type_ack_pdu_tag:(* = 0x60000002 *)
         (ack_pdu:    (*@T [APPLICATION 2] *) ^Ack_pdu_type);
       end; (* of type Npdu_type *)


               (* Primitives used for ASN.1-Estl *)
(* Predefined primitives for manipulating LIST structure *)
procedure C_ListCount(TheList: LIST_OF; var Count: integer); primitive;
       (* returns the list length (number of list items) of a list *)
procedure C_ListFirst(TheList: LIST_OF; var TheFirst: UNIV); primitive;
       (* returns the first item of the list in TheFirst and assigns
          the list "casn1_nextcurrent" field to the next item *)
procedure C_ListNext(TheList: LIST_OF; var Next: UNIV); primitive;
       (* returns the item pointed by "casn1_nextcurrent" in Next and
          assigns the "casn1_nextcurrent" field to the next item *)
procedure C_ListTrim(var TheList: LIST_OF; var LastRemoved: UNIV); primitive;
       (* remove the last list item from the list and
         returns the removed item value *)
procedure C_ListEOL(TheList: LIST_OF; var ENDOFLIST: boolean); primitive;
       (* returns TRUE if C_ListNext can not return any more items because
          end of list has been reached. Otherwise, it returns FALSE *)
procedure C_ListAdd(var TheList: LIST_OF; TheItem: UNIV); primitive;
       (* inserts a new list item into the list as the first list item *)
```

```
procedure C_ListAppend(var TheList: LIST_OF; TheItem: UNIV); primitive;
       (* inserts a new list item at the end of the list *)
procedure C_ListEmpty(TheList: LIST_OF; var Empty: boolean); primitive;
       (* returns TRUE if the list contains no items,
         returns FALSE otherwise. *)
procedure C_ListRemove(var TheList: LIST_OF); primitive;
       (* removes the last list item accessed by C_ListNext or C_ListFirst
         from the list. *)
procedure C_ListMerge(var Dest: LIST_OF; var Source: LIST_OF); primitive;
       (* merges two lists into one and Source-head is free. *)
procedure C_ListFree(var TheList: LIST_OF); primitive;
       (* frees the space occupied by a list. *)
procedure C_ListCopy(var Destination: LIST_OF; Source: LIST_OF);
            primitive;
       (* make a copy of a list copying complete structures. *)

(* Predefined primitives for manipulating OCTSTRING structure *)
procedure o_length(OS: OCTSTRING; var LEN: integer); primitive;
       (* return the actual length of OS variable *)
procedure o_null(var OS: OCTSTRING); primitive;
       (* initialize the variable OS to a null OCTSTRING *)
procedure o_copy(FROM_OS: OCTSTRING; var TO_OS: OCTSTRING); primitive;
       (* copy FROM_OS into TO_OS *)
procedure o_create(var OS: OCTSTRING; InitVal: DATA_TYPE); primitive;
       (* create OS given a DATA_TYPE InitVal *)
procedure o_get(OS: OCTSTRING; offset: integer; var value: OCTET); primitive;
       (* return the octet at the specified offset (0 is first) *)
procedure o_put(var OS: OCTSTRING; offset: integer; value: OCTET); primitive;
       (* put the octet at the specified offset (filler is '\0') *)
procedure o_assemble(var BASE: OCTSTRING; var ADDITION: OCTSTRING); primitive;
       (* append ADDITION to BASE; set ADDITION pointer to NULL *)
procedure o_fragment(var HEAD: OCTSTRING; var OLD: OCTSTRING; LEN: integer);
            primitive;
       (* OLD is fragmented as follows: LEN first octets are moved to HEAD;
            the tail remains in OLD *)
procedure o_free(var ToScrap: OCTSTRING); primitive;
       (* releases the space of OCTSTRING ToScrap *)

(* Predefined primitives for manipulating BITSTRING structure *)
procedure b_length(BS: BITSTRING; var LEN: integer); primitive;
       (* return the actual length of BS variable *)
procedure b_null(var BS: BITSTRING); primitive;
       (* initialize the variable BS to a null BITSTRING *)
procedure b_copy(FROM_BS: BITSTRING; var TO_BS: BITSTRING); primitive;
       (* copy FROM_BS into TO_BS *)
procedure b_create(var BS: BITSTRING; InitVal: DATA_TYPE); primitive;
       (* create BS given a DATA_TYPE InitVal *)
procedure b_get(BS: BITSTRING; offset: integer; var value: BIT); primitive;
       (* return the bit at the specified offset (0 is first) *)
procedure b_put(var BS: BITSTRING; offset: integer; value: BIT); primitive;
       (* put the bit at the specified offset (filler is 0) *)
procedure b_assemble(var BASE: BITSTRING; var ADDITION: BITSTRING); primitive;
       (* append ADDITION to BASE; set ADDITION pointer to NULL *)
procedure b_fragment(var HEAD: BITSTRING; var OLD: BITSTRING; LEN: integer);
            primitive;
       (* OLD is fragmented as follows: LEN first bits are moved to HEAD;
            the tail remains in OLD *)
procedure b_free(var ToScrap: BITSTRING); primitive;
       (* releases the space of BITSTRING ToScrap *)


       (* primitives for de/coding type U_Data_type *)
procedure C_encode_U_Data_type
       (var BER: ASN1_DATA_TYPE; var ToCode: U_Data_type); primitive;
procedure C_decode_U_Data_type
       (var CStruct: U_Data_type; var ToDecode: ASN1_DATA_TYPE); primitive;

       (* primitives for mapping Items in LIST of type IA5String *)
procedure PutItem_IA5String(var TheItem: UNIV; var ToMap: IA5String); primitive;
procedure GetItem_IA5String(var ToUnMap: IA5String; var TheItem: UNIV); primitive;

       (* primitive to duplicate/free U_Data_type: C_ListCopy/C_ListFree
          To initialize this list, you must use ListCreateU_Data_type *)
function ListCreateU_Data_type: LIST_OF; primitive;
```

```
       (* primitives for de/coding type Seq_type *)
procedure C_encode_Seq_type
       (var BER: ASN1_DATA_TYPE; var ToCode: Seq_type); primitive;
procedure C_decode_Seq_type
       (var CStruct: Seq_type; var ToDecode: ASN1_DATA_TYPE); primitive;

       (* primitives for de/coding type Data_pdu_type *)
procedure C_encode_Data_pdu_type
       (var BER: ASN1_DATA_TYPE; var ToCode: Data_pdu_type); primitive;
procedure C_decode_Data_pdu_type
       (var CStruct: Data_pdu_type; var ToDecode: ASN1_DATA_TYPE); primitive;

       (* primitives to duplicate/free Structure of type Data_pdu_type *)
procedure s_copyData_pdu_type
       (var ToStruct: Data_pdu_type; var FromStruct: Data_pdu_type); primitive;
procedure s_freeData_pdu_type
       (var ToFree: Data_pdu_type); primitive;

       (* primitives for de/coding type Ack_pdu_type *)
procedure C_encode_Ack_pdu_type
       (var BER: ASN1_DATA_TYPE; var ToCode: Ack_pdu_type); primitive;
procedure C_decode_Ack_pdu_type
       (var CStruct: Ack_pdu_type; var ToDecode: ASN1_DATA_TYPE); primitive;

       (* primitives to duplicate/free Structure of type Ack_pdu_type *)
procedure s_copyAck_pdu_type
       (var ToStruct: Ack_pdu_type; var FromStruct: Ack_pdu_type); primitive;
procedure s_freeAck_pdu_type
       (var ToFree: Ack_pdu_type); primitive;

       (* primitives for de/coding type Npdu_type *)
procedure C_encode_Npdu_type
       (var BER: ASN1_DATA_TYPE; var ToCode: Npdu_type); primitive;
procedure C_decode_Npdu_type
       (var CStruct: Npdu_type; var ToDecode: ASN1_DATA_TYPE); primitive;

       (* primitives to duplicate/free Structure of type Npdu_type *)
procedure s_copyNpdu_type
       (var ToStruct: Npdu_type; var FromStruct: Npdu_type); primitive;
procedure s_freeNpdu_type
       (var ToFree: Npdu_type); primitive;

(* INIT primitive for CASN1: should be called once at the beginning *)
procedure INITMEM; primitive;


-----Excerpts from FILE c_routine.c containing algorythmic body of primitives:
/* Files of routines generated by ASN1ESTL */
#include "/usr/aigle/protoco/include/estl/pascal.h"
#include "c_include.h"
#include "c_type.h"

       /* routine definition for PutItem/GetItem for type IA5String */
PutItem_IA5String(TheItem, ToMap)
 UNIV * TheItem;
 IA5String *   ToMap;
 { *TheItem = (UNIV) *ToMap; }
GetItem_IA5String(ToUnMap, TheItem)
 IA5String *   ToUnMap;
 UNIV * TheItem;
 { *ToUnMap = ((IA5String) *TheItem); }


       /* routine for initializing List U_Data_type */
LIST_OF ListCreateU_Data_type ()
{return(C_ListCreate(1, copyItemOCTS, freeItemOCTS)); }
/* copyItemOCTS and freeItemOCTS are the functions to copy/free items of
   this LIST; their function pointer is stored in a control block */


       /* routine s_copyData_pdu_type for duplicating Structure Data_pdu_type */
s_copyData_pdu_type(ToStruct, FromStruct)
 Data_pdu_type *       ToStruct;
 Data_pdu_type *       FromStruct;
 {if (ToStruct==NULL) {
```

```
       printf("\n\nERROR from Structure s_copyData_pdu_type: NULL pointer\n\n"); return; }
 bcopy(FromStruct, ToStruct, sizeof(Data_pdu_type));
 C_ListCopy(&ToStruct->ndata, FromStruct->ndata);
 }


       /* routine s_freeData_pdu_type for freeing Structure Data_pdu_type */
s_freeData_pdu_type(ToFree)
 Data_pdu_type *       ToFree;
 {if (ToFree==NULL) {
       printf("\n\nERROR from Structure s_freeData_pdu_type: NULL pointer\n\n"); return; }
 C_ListFree(&ToFree->ndata); }


       /* routine definition copyItemData_pdu_type for copying this List */
UNIV * copyItemData_pdu_type(AnItem)
 UNIV * AnItem;
 {Data_pdu_type *      temp = (Data_pdu_type *) malloc(sizeof(Data_pdu_type));
 s_copyData_pdu_type(temp, AnItem);
 return((UNIV *)temp); }

       /* routine definition for freeItemData_pdu_type in ListFree */
void freeItemData_pdu_type(AnItem)
 UNIV * AnItem;
 { s_freeData_pdu_type(AnItem); }


       /* routine definition for de/coding type Npdu_type */
IDX            *Encode_Npdu_type      ();
Npdu_type      *Decode_Npdu_type      ();

C_encode_Npdu_type(BER, ToCode)
 ASN1_DATA_TYPE *      BER;
 Npdu_type *   ToCode;
 { IDX  *temp;
 temp = Encode_Npdu_type(0, 0, 0, ToCode);
 BER->octetseq = (byte *) serIDX(temp, &BER->octetseq, &BER->length);
 }

C_decode_Npdu_type(CStruct, ToDecode)
 Npdu_type *   CStruct;
 ASN1_DATA_TYPE *      ToDecode;
 { IOP  temp;
 temp.b = ToDecode->octetseq;
 temp.f = NULL; temp.fname[0] = '\0';
 bzero(CStruct, sizeof(Npdu_type)); /* CHOICE type: init pointers to NULL */
 Decode_Npdu_type(0, 0, 0, &temp, CStruct);
 }


       /* routine s_copyNpdu_type for duplicating Structure Npdu_type */
s_copyNpdu_type(ToStruct, FromStruct)
 Npdu_type *   ToStruct;
 Npdu_type *   FromStruct;
 {if (ToStruct==NULL) {
       printf("\n\nERROR from Structure s_copyNpdu_type: NULL pointer\n\n"); return; }
 bzero(ToStruct, sizeof(Npdu_type));
 ToStruct->choice = FromStruct->choice;
 switch(FromStruct->choice) {
  case Npdu_type_data_pdu_tag: ToStruct->data_pdu = (Data_pdu_type *) malloc(sizeof(Data_pdu_type));
               s_copyData_pdu_type(ToStruct->data_pdu, FromStruct->data_pdu); break;
  case Npdu_type_ack_pdu_tag: ToStruct->ack_pdu = (Ack_pdu_type *) malloc(sizeof(Ack_pdu_type));
               s_copyAck_pdu_type(ToStruct->ack_pdu, FromStruct->ack_pdu); break;
  }
 }

       /* routine s_freeNpdu_type for freeing Structure Npdu_type */
s_freeNpdu_type(ToFree)
 Npdu_type *   ToFree;
 {if (ToFree==NULL) {
       printf("\n\nERROR from Structure s_freeNpdu_type: NULL pointer\n\n"); return; }
 switch(ToFree->choice) {
  case Npdu_type_data_pdu_tag: s_freeData_pdu_type(ToFree->data_pdu);
               free(ToFree->data_pdu); break;
  case Npdu_type_ack_pdu_tag: s_freeAck_pdu_type(ToFree->ack_pdu);
```

```
              free(ToFree->ack_pdu); break;
  }
 }


=========================================================================
Annex-2: Alternating Bit Protocol Specification.

Specification using PDU definition and primitives from Annex-1:

Specification Example systemprocess; timescale seconds;
    { This is the top level module body (specification)
      The specification has the attribute systemprocess.
      The time scale for delays is in seconds.                }


{ ASN.1 const: }
#include asn1estl/const.e
{ any base-type is used to specify that an implementer must
               define these constants for his environment. }
Retran_time = any integer; { retransmission time }


{ ASN.1 types: }
#include asn1estl/new-ab.asn1.e


{ Channel definitions for communication between the processes }

channel U_access_point(User,Provider);
    by User:
        SEND_request (Udata:  U_Data_type);
        RECEIVE_request;
    by Provider:
        RECEIVE_response(Udata:  U_Data_type);

channel CODING_access_point(User, Provider);
    by User:
       PDU_send(PDU: Npdu_type);
    by Provider:
       PDU_receive(PDU: Npdu_type);

channel N_access_point(User,Provider);
    by User:
        DATA_request(ASN1_coded_PDU: ASN1_DATA_TYPE);
    by Provider:
        DATA_response(ASN1_coded_PDU: ASN1_DATA_TYPE);

{ Module header definitions }


module Alternating_bit_type process;
           ip {interaction point list }
             U:  U_access_point(Provider) individual queue;
             C:  CODING_access_point(User) individual queue;
end; { of module header definition }

    { The module has two interaction points named U and N;
      the roles of the module are named:
          Provider with respect to U, and
          User with respect to C.          }

{ The body for alternating bit is defined below: }

body Alternating_bit_body for Alternating_bit_type;

type
    Buffer_type =
        record        { record introduces a data structure }
            buffer:  Data_pdu_type;
            empty:   boolean
        end;

var
    Send_buffer, Recv_buffer: Buffer_type;
```

```
    Send_seq,Recv_seq: Seq_type;
    temp_data: Data_pdu_type;
    temp_ack:  Ack_pdu_type;
    B: Npdu_type;
    transform: UNIV;
    Str: IA5String;

state ACK_WAIT, ESTAB; { state definition part }

stateset     { state-set-definition-part }
 EITHER  =   [ACK_WAIT, ESTAB];

procedure Format_data(Msg: Data_pdu_type; var B: Npdu_type);
 begin
     B.choice :=  Npdu_type_data_pdu_tag;
     new(B.data_pdu);
     s_copyData_pdu_type(B.data_pdu^, Msg);
 end;

procedure Format_ack (Msg: Ack_pdu_type; var B: Npdu_type);
 begin
     B.choice :=  Npdu_type_ack_pdu_tag;
     new(B.ack_pdu);
     s_copyAck_pdu_type(B.ack_pdu^, Msg);
 end;


{ two variables of type buffer_type are used
       to hold messages ( of type data_pdu_type):
   Send_buffer for sending, Recv_buffer for receiving.
   The following procedure and functions are used
   manipulate buffer_type variables }

procedure Empty_buf(var Buf: Buffer_type);
begin
  with Buf do
    if not empty then
      begin
      empty := true;
      C_ListFree(buffer.ndata); { free space occupied by SET OF }
      end
end;

procedure Store(var Buf: Buffer_type; Msg: Data_pdu_type);
begin
  with Buf do
    begin
    empty := false;
    { in this case a copy of the data is made }
    s_copyData_pdu_type(buffer, Msg);
    end
end;

procedure Retrieve(Buf:  Buffer_type; var Msg: Data_pdu_type);
begin
  s_copyData_pdu_type(Msg, Buf.buffer);
end;

procedure Inc_send_seq;
 begin
     Send_seq := (Send_seq + 1) mod 2
 end;


procedure Inc_recv_seq;
 begin
     Recv_seq := (Recv_seq + 1) mod 2
 end;

initialize { initialization-part of the alternating bit process }

to ESTAB  { initialize major state variable to ESTAB }
   begin  { initialize variables }
        Send_seq := 0;
        Recv_seq := 0;
```

```
        Send_buffer.empty := true;
        Recv_buffer.empty := true;
   end;

trans {  transition-declaration-part of the alternating bit process   }

from ESTAB
    to ACK_WAIT
        when U.SEND_request
         begin
             temp_data.seq := Send_seq;  { temp_data holds the sending seq num }
             temp_data.ndata := Udata;   { and User Data }
             C_ListFirst(Udata, transform);
             GetItem_IA5String(Str, transform);
             writeln('\nSending Data. The first Item of the list is:\n\t%s\n',
                      Str^.data);
             Store(Send_buffer, temp_data);{ store temp_data in sending buffer }
             Format_data(temp_data, B);  { format a network message }
             output C.PDU_send(B);
            C_ListFree(Udata);  { releases space of arrived user data }
         end;

from ESTAB
    to same
        when C.PDU_receive
            provided PDU.choice=Npdu_type_ack_pdu_tag
            begin
               { purge the queue to prevent deadlock }
             dispose(PDU.ack_pdu);
            end;

from EITHER
    to same
        when U.RECEIVE_request
            provided not Recv_buffer.empty
             begin
             Retrieve(Recv_buffer, temp_data); { retrieve received message }
             output U.RECEIVE_response(temp_data.ndata);
             Empty_Buf(Recv_buffer)     { remove message from receiving buffer }
             end;

from ACK_WAIT
    to ACK_WAIT
        delay (Retran_time)
         begin
             Retrieve(Send_buffer, temp_data);
                                  { retrieve message to be retransmitted }
             Format_data(temp_data, B); { format a network message }
             output C.PDU_send(B);
         end;


from ACK_WAIT
    to ESTAB
       when C.PDU_receive
         provided (PDU.choice=Npdu_type_ack_pdu_tag) and
                  (PDU.ack_pdu^.seq=Send_seq) { ACK is OK }
         begin
             Empty_buf(Send_buffer);    { remove acknowledged message }
             dispose(PDU.ack_pdu);
             Inc_send_seq
         end;

from ACK_WAIT
    to same
       when C.PDU_receive
         provided (PDU.choice=Npdu_type_ack_pdu_tag) and
                  (PDU.ack_pdu^.seq<>Send_seq) { ACK is bad }
        begin
            { to prevent a livelock: a USEND_request was sent
              and the alternating_bit instance receives a bad ack
              C.DATA_response ack.                              }
            dispose(PDU.ack_pdu);
        end;
```

```
from EITHER
    to same
       when C.PDU_receive
            provided PDU.choice=Npdu_type_data_pdu_tag
             begin
               temp_ack.seq := PDU.data_pdu^.seq;
               Format_ack(temp_ack, B);
               output C.PDU_send(B);
               if PDU.data_pdu^.seq = Recv_seq then
                 begin
                     Store(Recv_buffer,PDU.data_pdu^);
                     Inc_recv_seq
                 end;
               C_ListFree(PDU.data_pdu^.ndata);
               dispose(PDU.data_pdu);
            end;

end; {  of the Alternating_bit_body}



module CODING_type process;
       ip {interaction point list }
         Protocol:  CODING_access_point(Provider) individual queue;
         N:         N_access_point(User)     individual queue;
end; { of module header definition }


body CODING_body for CODING_type;

initialize
  begin
  INITMEM; { to initialize CASN1 tool *)
  end;

trans
  when Protocol.PDU_send
    var ASN1_coded_PDU: ASN1_DATA_TYPE;
    begin
    C_encode_Npdu_type(ASN1_coded_PDU, PDU);
    output N.DATA_request(ASN1_coded_PDU);
    s_freeNpdu_type(PDU);
    end;

trans
  when N.DATA_response
    var decoded_PDU: Npdu_type;
    begin
    C_Decode_Npdu_type(decoded_PDU, ASN1_coded_PDU);
    dispose(ASN1_coded_PDU.octetseq);
    output Protocol.PDU_receive(decoded_PDU);
    end;

end; { of body CODING_body }


modvar
    { module-variable-declaration-part of the specification }

    Alternating_bit: Alternating_bit_type;
    CODING: CODING_type;

initialize { initialization-part of the specification }

 begin { module initialization }
   init Alternating_bit with
        Alternating_bit_body;
   init CODING with
        CODING_body;
   connect Alternating_bit.C to CODING.Protocol;
 end; { of module initialization within the
        specification's initialization-part }

end.  { End of specification; the specification has no transition part }
```