ESTELLE: A FORMAL DESCRIPTION TECHNIQUE FOR COMMUNICATION PROTOCOLS

Michel Barbeau, VE2BPM
3360 Marechal, app. 305
Montreal, Canada
H3T 1M9

1. Introduction

A communication protocol is a set of rules for data exchange between entities of a computer network. A communication protocol may be defined, or specified, by text written in a natural language such as english. Specifications of this type are sometimes ambiguous and imprecise, mainly because today's communication protocols are very complex. Each reader makes his own assumptions and interpretations of the unclear aspects. The corresponding implementations, or concrete realizations, of the protocol, made by different groups of people, may have incompatible behaviours under certain circumstances and therefore they cannot always work together properly. In order to specify unambiguously, clearly and concisely communication protocols, what are called, Formal Description Techniques (FDT) have been developed.

Estelle [1] is a FDT developed by ISO (International Standard Organization). It is based on an Extended Finite State Machine model (EFSM). A finite state machine is a simple abstract device which has states and transitions labelled by input and output symbols. A state transition table is one of the possible ways to represent textually a FSM. FSM's are frequently used to model the control flow of systems. Communication softwares, such as data link protocols, have usually a component that can be represented by FSM's. As an example see the state transition tables of the AX.25 link-layer protocol in appendix D of [2]. Protocols have also data flow aspects involving interaction parameters, different kinds of variables and data operations. The data flow aspects are hard to represent using only FSM's. Estelle extend the idea of FSM in a sense that variables, actions and predicates, operating on those variables, are added to this basic model. The syntax of Estelle has been defined from the syntax of the programming language Pascal. New elements have been added in order to make easier the definition of aspects particular to communication protocols. In Estelle can be expressed both the control flow aspect, using FSM's, and the data flow aspect, using Pascal's elements, of a communication protocol. Protocol specifications written in the formal language Estelle are said formal with respect to informal specifications written in a natural language such as English.

Section two introduces the alternating bit, a simple data transfer communication protocol. Then, in section three, the alternating bit is used to present the FDT Estelle for the specification of communication protocols.

2. The Alternating Bit Protocol

The alternating bit is a very simple data transfer protocol which can be used, as **the** AX.243 packet-radio protocol [2], in the data link layer of the ISO reference model. This section gives a short introduction to the alternating bit.

The protocol comprises a mechanism giving the capability to recover data losses during the data transfer. It prevents also the transmitter from overloading the receiver with data. Two kinds of data blocks, or Protocol Data Units (PDU's), are exchanged between entities that are using the rules of the alternating bit. The first kind of PDU is named DT. A DT block is composed of two fields: a sequence number field and a user's data field. The second kind of PDU is named AK. AK blocks are used to acknowledge the successful transfer of DT blocks from the transmitter to the receiver. An AK PDU contains the sequence number of an acknowledged DT PDU. Values of sequence numbers alternate between zero and one. The interaction with the users of the protocol is represented by the messages SEND_request, RECEIVE_request and RECEIVE_response. The interaction RECEIVE_request has no parameter but the messages SEND_request and RECEIVE_response have both a parameter for user's data.

Figure 1 shows a typical dialogue between two entities that are using the alternating bit. The transmitter initiates the dialogue by a transmission request (1). The protocol generates a DT, numbered zero. This DT is correctly received by the peer entity (2). The user's request for data is satisfied by a RECEIVE_response which conveys the value took from the data field of the DT PDU. An acknowledgement is also sent to the transmitter. Only one DT PDU may be transmitted at once. Its successful reception must be acknowledged before transmitting more data.

Losses of PDU's are recovered. **A** second interaction SEND-request (3) generates a DT PDU numbered one. Its corresponding acknowledgement is lost during the transfer (4). After a timeout period the emitter retransmits the same DT block (5). Its reception is now

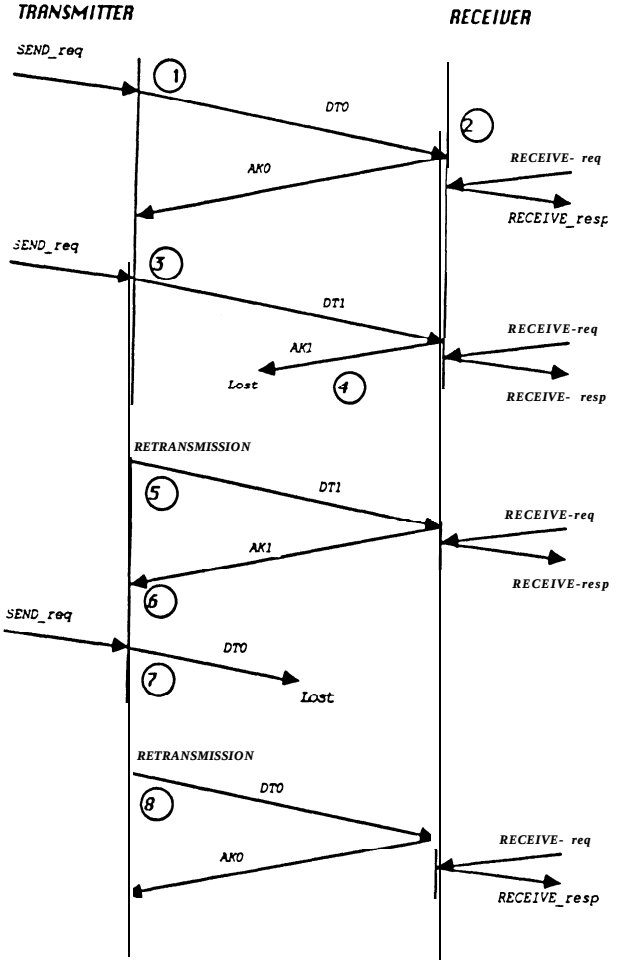

Fig. 1 Data transfer with the alternating bit

acknowledged (6). In the case where the DT block itself is lost **(7)** retransmission also occurs after a timeout period (8).

3. The Formal Specification of a Protocol

The language Estelle is used to define formally the behaviour to which, implementations of a communication protocol, might conform. However, a certain level of abstraction is kept with respect to the specific characteristics of some particular implementation. A protocol description in Estelle is generally composed of two main parts:

- The first part contains descriptions of channel types through which the protocol will exchange messages **with.** its environment (i.e. the users and the communication services provided by the layer beneath). Channels are defined in terms of the messages that can be sent by each entity communicating through it. Messages can have parameters, therefore they are defined along with their parameter identifiers and their respective data types.

- The protocol description itself is structured into one or several cooperating modules, defined in the second part of the specification. Each module corresponds to an EFSM (i.e. a FSM capable of having memory). A module may contain data type, variable and procedure declarations. Each module has interaction points, each one associated to a channel type. Communications with other modules and the protocol's environment take place at those interaction points. The control and data flow aspect of a module are essentially described by a group of transitions. **A** complete module specification consists of a module header, where its interaction points are defined, and generally **qne** module body, where data flow and control aspects are defined.

The syntax of the Estelle language is essentially an extension of the syntax of the Pascal language. New syntactic elements have been added to provide the facilities to define aspects which are specific to communication protocols. Appendix A shows the formal specification in Estelle of the alternating **bit protocol, presented in section** 2. The following is a discussion of this specification.

3.1 Channels **and Interactions**

At the beginning of the specification is declared the Pascal record *Ndata_type* describing the structure of the PDU kinds exchanged by the protocol implementations. The alternating bit uses two kinds of PDU. The fields *Id* identifies the PDU's. For a DT PDU the fields *Data* snd *Seq* are used. For an AK the field *Seq* only is used. This definition does not take into account the encoding of the PDU's in terms of bits and bytes. This information would be given in an informal specification (i.e. using a natural language). Therefore, formal and informal techniques complete each other.

Channel statements introduce the types of inter= action exchanged by the involved entities. Communication between the user of the alternating bit and the protocol will take place **over** a channel of type *U_access_point* and over a channel of type *N_access_point* between the protocol and the layer beneath (i.e. the physical layer). Over a channel of **type** *U_access_point*

the user can send the interactions *SEND_request* and *RECEIVE_request*. The protocol may, over the same channel, send the message *RECEIVE_response*. Names and data types of interaction parameters are specified in parentheses following each message identifier. Interactions such as *SEND_request*, *RECEIVE_request* and *RECEIVE_response* are generally called service primitives. The channel *N_access_point* can be interpreted in a similar fashion. Those channel declarations do not suggest any concrete realization. Services primitives may be implemented as procedure calls, subroutines linkages or interprocess communication.

3.2 Modules

Modules are defined using the *module* and *body* statements. A *module* statement is used to define what is called the module header. The header names the points of interaction with the module's environment, the channel type used at each of these points and the role attributed to the module. The module header *Alternating_bit_type* 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 N.

To each header type, are associated one or several module bodies. This way, to the same external structure may correspond different internal behaviours. The definition of each of these bodies begins with the keyword *body*. The specification of appendix A is structured into a single module. To the module header *Alternating_bit_type* corresponds a single module body identified *Alternating_bit_body*. Generally specifications of complex protocols are structured into many modules and those modules may themselves embed submodule declarations.

The skeleton of a module body is a finite state machine. Because it is difficult and generally impossible to specify a fairly complex protocol using only FSM's, elements of the Pascal language have been added to extend this basic concept. Usually a module body comprises three consecutive major sections: a declaration part, an initialization part and a transition part.

Any kind of declaration that could be found in Pascal programs may be used in an Estelle specification. The body *Alternating_bit_body* contains declarations of constants, data types, variables, procedures and functions.

The representation of the data type *Buffer_type* is undefined. Instead the definition has been replaced by the symbol "...". The body of some procedures and functions is defined as *external* or *primitive*. Those undefined aspects of the specification are left to the protocol implementers. They will choose themselves the more suitable representation and the algorithms to manage data buffers.

The state of a module, during its execution, is characterized by the values of its variables. One of these, the variable state, plays a key function. It will save the major state name (i.e. either ACK_WAIT or ESTAB in this case). This variable corresponds to the FSM component of the module. The other variables save sequence numbers and data elements, they are called context variables.

The initial state of a module is defined in the initialization part. First, this part specifies the initial major state name using the statement, *to*. Then, assignment statements give initial values to the context variables.

3.3 Transitions

The transitions define the potential state changes of a module. The module *Alternating_bit_body* has five transition types. The first three are related to data transmission. The last two handle the reception of data. Each transition has a *begin-end* bloc, similar to a Pascal procedure. This bloc is preceded by a sequence of clauses specific to Estelle. The clauses *from* and *to* express the major state change made when the transition is executed. The firing of the transition depends of the current major state and also on the truth value of a predicate specified in the *provided* clause. This predicate contains references to context variables. The firing may also depend on the arrival of an input interaction which is selected by the clause *when*, otherwise the transition is called spontaneous.

In brief, a module makes a transition if one of its transitions is enabled. A transition is enabled if the conditions jointly specified by the clauses *from*, *when* and *provided* are satisfied. The transition thru a state, first specified by the clause *to* then by the new values assigned to the context variables in the *begin-end* bloc, will be made. Any Pascal statement may be used in the *begin- end* bloc. Moreover, output messages can be generated using *output* statements.

A message sent by an output interaction is put in a queue associated to its receiver. The receiver will retrieve the message from the queue when he will be able to process it. With the statement *stateset* may be declared sets of major state names. References to these state sets can be made in the *from* clauses.

Estelle contains also statements for dynamic management of modules instances (i.e. creation, destruction, etc.). Moreover, Estelle comprises to whole standard

Pascal language. For a complete coverage of Estelle see [1].

4. Conclusion

FDT's are the first step toward the development of well defined and reliable communication softwares. FDT's provide clear and precise definition of communication protocols to implementers. Estelle is a FDT developed by IS0 that is expected to be extensively used in the future.

References

[1] ISO/TC 97/SC 21, *Estelle - A Formal Description Technique Based on an Extended State Transition Model*, DP 9074, 1986.

[2] FOX L. Terry, *AX.25 Amateur Packet-Radio Link-Layer Protocol*, ARRL, 1984.

Appendix **A: An Example Protocol Specification**

This specification has been extracted from reference [1]. Comments are placed, as in Pascal, in curly brackets. Note that this protocol definition may contain deadlocks.

```
specification Altbit;

type
    U-Data-type = . . . . { user data }
    Seq-type = 0..1 ; { sequence number range }
    Id-type = (DT, AK);
    Ndata_type = record
        Id : Id-type; { type of message }
        Data : U-Data-type; { user data }
        Sey : Seq_type; { sequence number }
    end;

{ Channel definitions }
channel U_access_point( User,Provider);
    by User:
        SEND-request (Udata : U-Data-type);
        REC'EIVE_request;
    by Provider:
        REC'EIVE_response( Udata : U-Data-type);

channel N_access_point(User,Provider);
    by User:
        DATA_request( Ndata : Ndata_type);
    by Provider:
        DATA_response( Ndata : Ndata-type);

{ Module header definition }
module Alternating-bit-type process;
    ip { interaction point list }
        U : U_access_point( Provider) common queue;
        N : N_access_point( User) individual queue
    end;

{ Module body definition }
body Alternating_bit_body for Alternating-bit-type;

const
    Retran_time = ...;
    { Retransmission time in seconds
    (defined by implementer) }

type
    Msg_type = record
        Msgdata : U-Data-type;
        Msgseq : Seq_type
    end;

    Buffer-type = . . . .

var
    Send-buffer, Recv_buffer : Buffer-type;
    Sendseq, Recvseq : Seq-type;
    P, Q : Msg-type;
    B : Ndata-type;

    state ACK-WAIT, ESTAB;

stateset
    EITHER = [ACK-WAIT, ESTAB];

procedure Copy(
    var To_Data:U_Data_type; From_data:U_Data_type);
external;
{ procedure provided by implementer:
copy a user data variable }

procedure Empty( var Data:U_Data_type);
primitive;
{ procedure provided by implementer:
initialize a variable holding user data
to the value no user data }

procedure Formatdata( Msg:Msg_type; var B:Ndata_type);
begin
    B.Id := DT;
    copy( B.Data, Msg.Msgdata);
    B.Seq := Msg.Msgseq;
end;
```

```
procedure Format_ack( Msg:Msg_type; var B:Ndata_type);
begin
    B.Id := AK;
    B.Seq := Msg.Msgseq;
end;

procedure Empty_buf( var Buf:Buffer_type);
primitive;
{ procedure provided by implementer:
set a buffer to empty }

procedure Store( var Buf:Buffer_type; Msg:Msg_type);
primitive;
{ procedure provided by implementer:
store a message into a buffer_type variable such that the
messages can be retrieved or removed in a FIFO manner }

procedure Remove( var Buf:Buffer_type);
primitive;
{ procedure provided by implementer:
remove the first message }

function Retrieve( Buf:Buffer_type):Msg_type;
primitive;
{ function provided by implementer:
retrieve the first message and return it;
the message is not removed }

function buffer_empty(Buf:Buffer_type):boolean;
primitive;
{ function provided by implementer:
check if a buffer contains a message }

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

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

{ initialization-part }
initialize
to ESTAB { initialize major state variable to ESTAB }
begin
    { initialize context variables }
    Send_seq := 0;
    Recv_seq := 0;
    Empty_buf(Send_buffer);
    Empty_buf(Recv_buffer);
end;

{ Transition part }
{ Sending data }
trans
from ESTAB to ACK_WAIT
when U.Send_request
begin
    copy(P.Msgdata,Udata);
    P.Msgseq := Send_seq;
    Store(Send_buffer,P);
    Format_data( P,B);
    output N.DATA_request(B);
end;

trans
from ACK_WAIT to ACK_WAIT
delay (Retran_time) { timeout }
begin
    P := Retrieve( Send_buffer);
    Format_data( P,B);
    output N.DATA_request( B);
end;

trans
from ACK_WAIT to ESTA:B
when N.DATA_response
provided (Ndata.Id = AK) and (Ndata.Seq = Send_seq)
begin
    { remove acknowledged message }
    Remove(Send_buffer);
    Inc_send_seq
end;

{ Receiving data }
trans
from EITHER to same
when N.DATA_response
provided Ndata.Id = DT
begin
    copy(Q.Msgdata, Ndata.Data);
    Q.Msgseq := Ndata.Seq;
    Format_ack(Q,B);
    output N.DATA_request(B);
    if Ndata.Seq = Recv_seq then
    begin
        Store(Recv_buffer,Q);
        Inc_recv_seq
    end
end;
```

```
trans
from EITHER to same
when U.RECEIVE_request
provided not buffer-empty( Recv_buffer)
begin
    { retrieve received message }
    Q := Retrieve( Recv_buffer);
    output U.RECEIVE_response( Q.Msgdata);
    { remove message from receiving buffer }
    Remove(Recv_buffer)
end;
end; { of module body }
end. { of specification }
```