Logical Verification and Implementation of Protocols
G.V. Bochmann
Proceedings of the Fourth Data Communications Symposium, Quebec, Canada, 1975
Pages 8-5 — 8-20

5.1

Abstract

The implementation and logical verification of communication protocols are considered in view of obtaining reliable communication systems. It is proposed that methods for specifying communication protocols should be useful for obtaining a comprehensive description, and for simplifying the logical verification and implementation of the specified protocol. These issues are discussed for a particular protocol, which is taken as an example throughout the paper. The example chosen is a simple HDLC protocol for two-way simultaneous data communication over a point-to-point data link. After an informal description, its operation is defined precisely. From the definition of the protocol, certain properties are proved which show the correct operation of the protocol. In particular, it is shown that line transmission errors are correctly recovered, and that the sequence counts can be represented modulo a given constant. A condition for completed data transmission is also given. The method for proving these properties is general, and can also be applied in the case of the more complex protocols that are used in actual applications. The second part of the paper deals with the problems of parallel processes and the implementation of protocols. Well structured high-level languages which include facilities for describing parallel processes and monitors are proposed as tools for the implementation of protocols. As an example, it is shown how such a language can be used to program the protocol introduced earlier. It is pointed out that this program can be easily obtained from the original protocol definition by doing only few changes and refinements. Therefore it is expected that such an implementation is relatively reliable.

1. Introduction

The design and implementation of appropriate communication protocols is an important part of the development of any new communication or distributed computer system. Normally, a protocol is designed to operate efficiently in normal situations, and also to deal with occasional erroneous behavior or malfunction of subsystems. The different situations a communication protocol has to cope with, are well described in reference 1. A realistic scheme of interprocess communications must foresee appropriate actions for the following situations :

(a) Normal communication between subsystems.
(b) Occasional erroneous behavior of one or several of the subsystems. This includes the occasional malfunction of the communication line. The recovery procedure will normally consist of trying the same action again.
(c) Long range erroneous behavior or failure of one or several subsystems. The recovery procedure will normally consist of reconfiguration of the system.

The design of a communication protocol is a difficult task because many different situations must be considered. Besides the normal operation of the protocol, situations due to erroneous behavior of one or several subsystems must be described. The synchronization between the subsystems in these situations is not very simple to understand. Therefore a logical verification of the protocol would be very useful. By logical verification we mean proving certain properties of the protocol which assure its correct operation in all situations. After its design, a protocol must be implemented in software and/or hardware. An implementation must be easy to understand, correct and efficient. The choice of a good programming language for the implementation is of large importance.

The method used for specifying protocols has a strong influence on how easy or difficult it is to design and implement a protocol for a given communication problem. We believe that the design, logical verification, and implementation of a protocol are interrelated. Therefore a method for specifying protocols should be useful for all three of these activities. More precisely, a method for specifying protocols should have the properties that :

(1) a protocol can be specified in a comprehensive form; in particular, the complete definition of a protocol can be partitionned into different levels of abstraction;
(2) the specification of a protocol allows proving certain properties of the protocol and its operation, proving in particular that the error recovery is effective, and that all possible situations of erroneous behavior have actually been considered;
(3) given the specification of a protocol, its implementation is simple, and part of the implementation may be obtained automatically.

It seems that at present, there is no method for specifying communication protocols that satisfies all these requirements. However, several tools for specifying, proving properties of, and implementing systems of parallel processes have been discussed in literature. The purpose of this paper is to show by an example that certain tools that have been developped in different fields of computer science can be useful for the design and implementation of communication protocols. We discuss in particular the possibility of logical verification by proving certain properties of protocols, and implementing protocols using a high-level language for parallel processes. Logical verification of protocols is useful for obtaining reliable communication systems, because it helps detecting weak points or errors in the protocol design which are difficult to find by simulation or testing. The use of a well-structured high-level language for the implementation of protocols has many advantages. It simplifies the programming effort, facilitates the detection of programming errors, makes the system more transportable, and clarifies the program documentation.

In this paper, we use a simple protocol for two-way simultaneous data communication as an example. This protocol is informally introduced in section 2. In section 3, we give a formal specification of the same protocol. This specification is then used to derive certain properties of the protocol and its operation. In particular, we show that the protocol correctly transmits message sequences, and that the internal sequence counts can be represented modulo a certain constant, thereby allowing a reasonable implementation. We use the technique of assertions [2] for verifying these properties. We note that this technique cannot be used for proving the absence of undesired loops, and the effective termination of a transmission sequence. These problems, however, have been considered in reference 4.

In section 4, we discuss how the concepts of parallel processes and monitors [5,6] can be used for the implementation of a protocol in a high-level programming language. As an example, we give the protocol introduced in section 2 in the form of a program which includes several processes and a monitor. Using a high-level language for the implementation of the system facilitates the use of structured programming. However, an implementation of such a language must be available.

2. A simple HDLC protocol

The following simple protocol will be used throughout this paper as an example. It is a protocol for two-way simultaneous data communication over a point to point data link. It is based on the HDLC (double numbering) standard [7], which is for instance adopted by IBM [8], and the canadian Datapac service [9]. We discuss in this paper only those aspects of the protocol which deal with message numbering and retransmission for error recovery. Other aspects, such as message formats, are described elsewhere [7]. The protocol is very simple, and uses only information format frames. It is supposed to be initialized by a line setup procedure not described here.

Each information frame sent over the line contains in its control field the send sequence count S, and the receive sequence count R of the transmitting station. Each received frame is verified by a redundancy check. The receiving station then compares the sequence count S of the frame with its own internal receive sequence counter R. If S is the next sequence to be received the information field of the frame is passed on to the user and the internal sequence counter R is incremented by one. Otherwise the received frame is ignored.

The reception of a frame with the receive sequence count R acknowledges all frames, transmitted in the opposite direction, with a send sequence count smaller or equal to R. Normally, messages are sent in sequence, and the send sequence counter S of the station is incremented by one for each frame sent. Retransmission of frames is initiated after a certain number of frames have been sent without being acknowledged. When the number of outstanding frames, i.e. frames sent but not yet acknowledged, becomes larger than a certain system constant M, the station transmits in sequence all frames following the last frame acknowledged.

For synchronizing the data link control with the speeds of the information source and sink the following procedure is adopted: (a) as long as a station has not obtained any more information to be transmitted it retransmits, in regular time intervals, the last frame; (b) if a station receives correctly a frame in sequence, but the information sink (i.e. the user or the user buffer) is not ready to accept the information contained in the frame the received frame is ignored; like a frame out of sequence.

So far, we have described information transfer from a source to a sink (see figure 1, upper half). However, the protocol supports two-way simultaneous data communication as indicated in figure 1. The stations at both sides of the data link have the same structure. Each contains a sending and a receiving part. For distinguishing the two sides of the data link, the elements of the station on the opposite side (i.e. on the right) are written with an overlining bar.

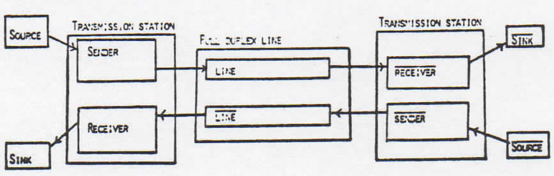

Figure 1: Two-way simultaneous data communication

3. A proof of correct operation

The section above gives an informal description of a simple HDLC protocol. In the following we show how certain useful properties can be proved about its operation. In order to do this, we first give a more formal definition of the protocol. This definition also serves as starting point for obtaining an implementation of the protocol, as discussed in section 4. From the definition of the protocol, we prove certain assertions about the states through which the protocol can pass during its operation. These assertions are then used to prove that the protocol correctly recovers the transmission errors of the data link, and that the sequence numbers can be represented modulo the constant 2M.

3.1 Definition of the protocol

The information to be transmitted from a source to a sink consists of a sequence of messages. Each message is transmitted as the information field of a single frame (i.e. transmission bloc). We write the i-th frame that travels on the transmission line in the form $<S_i, R_i, m_i>$, where S_i and R_i are the send and receive sequence counts, and m_i is the information field, i.e. a message; all other fields of the frame (including the redundancy check field) can be ignored for our present purposes. At a given instant, during the operation of the protocol, a certain number of frames travel on each line. This sequence of frames is written as:

$$<S_\ell, R_\ell, m_\ell> \cdots <S_{i+1}, R_{i+1}, m_{i+1}> <S_i, R_i, m_i> \cdots$$
$$\cdots <S_n, R_n, m_n>$$

where $<S_\ell, R_\ell, m_\ell>$ is the last frame sent by the transmitting station, and $<S_n, R_n, m_n>$ is the next frame to be received by the receiving station.

Each station contains a message buffer *send-buffer* for the messages to be transmitted. For simplicity we suppose that its size is unlimited; later, however, we show that a cyclic buffer for 2M messages would be sufficient. Each station also contains the internal sequence counters L, A, S, N, and R. The meaning of the values of these counters is the following:

- L is the sequence count of the last message obtained from the source.
- A is the highest sequence for which a correct reception, at the opposite station, has been acknowledged.
- S is the sequence count of the last frame sent.
- N is the highest value that has been reached by S so far.
- R is the sequence count of the last received frame that has been passed on to the sink. We note that this counter refers to the transmission in the opposite direction.

At this point, these meanings have to be understood as pure comments which may help the reader to understand the protocol definition that follows. Based on the definition, we shall show later (in section 3.3) that the counter values actually have these meanings.

The state of the protocol, at a given instant during its operation, is defined by the values S, R, A, N, and L of the internal counters of one station, the values $\bar{S}$, $\bar{R}$, $\bar{A}$, $\bar{N}$, and $\bar{L}$ of the counters of the opposite station, and the sequences of frames that travel on the transmission lines. The initial state of the protocol is characterized by

$$S = R = A = N = L = 0 \quad (\text{and } \bar{S} = \bar{R} = \bar{A} = \bar{N} = \bar{L} = 0)$$

and no frames on the lines. The reader can find in figure 2 a typical situation of a transmission station during the operation of the protocol.

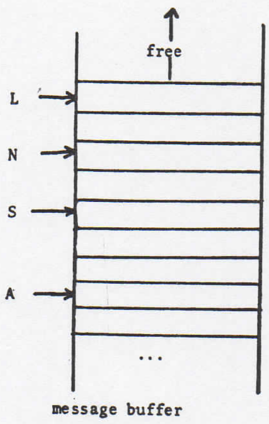

Figure 2: Typical situation of the counter values and the message buffer of a transmission station (during retransmission).

State transitions can occur due to the events of sending or receiving a frame, or obtaining a new message from the source. The state transformation of each of these events, which exclude one another in time, is described by the statements below.

We note that $M \geq 1$ is a system constant which determines after how many outstanding non-acknowledged frames retransmission is to start. Normally this constant will be adjusted to the expected delay for receiving an acknowledgement for a given frame, and the number of frames sent in unit time. The statements of the events are written in a free style Pascal, and comments are given in brackets { }.

(a) <u>The event of obtaining a new message from the source</u>

```
(1)  obtain (message);
(2)  L := L + 1;
(3)  send-buffer [ L ] := message;
```

(b) <u>The event of sending a frame</u>

```
(4)  if S ≥ L
          {no new message to be transmitted}
(5)       or S ≥ A + M
          {too many outstanding frames}
(6)  then if A = L then S:= A
                          {retransmit last frame}
(7)            else S:= A + 1
         {retransmit last non-acknowledged frame}
(8)  else S := S + 1;
          {send next frame}
(9)  N:= max (N,S);
          {increment N if necessary}
(10) send-frame ( <S,R, send-buffer [ S ]> );
```

(c) <u>The event of receiving a frame</u>

```
(11) listen-to-next-frame
     ( <S_frame, R_frame, message> );

(12) if redundancy-check-is-valid
(13) then begin
(14)      if R_frame > A then A:= R_frame;
                              {adjust A}
(15)      if S_frame = R + 1
                       {correct sequence}
(16)         and sink-is-ready-to-receive
(17)      then begin
(18)         R:= R + 1;
(19)         pass-on-to-sink (message)
(20)         end
(21) end
```

Corresponding events can also occur at the opposite station. The order in which these events occur is not specified, however, they never occur simultaneously (mutual exclusion). We assume that there will always be a successive sending and receiving event after a finite time interval, i.e. the operation of the protocol will never stop.

3.2 <u>Properties of the protocol</u>

The protocol defined above transmits messages in correct sequence from source to sink simultaneously in both directions, and it recovers line transmission errors, as long as the line does not break down permanently. This can be summarized by the following properties, which hold for each of the two transmission stations at any instant between the occurrences of the events (a), (b), or (c) defined above, and which are proved below.

<u>Property (1)</u>: At any given instant, exactly R messages, from the source at the opposite station, have been passed on to the sink correctly (i.e. without any bit error, and in the correct sequence).

<u>Property (2)</u>: At any given instant, at least A messages from the source have been correctly passed on to the sink at the opposite station.

<u>Property (3)</u>: When $A = L$, all messages obtained from the source (and no more) have been correctly passed on to the sink at the opposite station.

We note that the assertion of correct transmission relies on the assumption that the redundancy check on received frames detects all transmission bit errors.

The properties (1) and (3) show the "correct operation" of the protocol. The following property (4) is important for an implementation of the protocol. We have so far assumed that the sequence counts can grow indefinitely. We see now that a representation modulo 2M can be used:

<u>Property (4)</u>: The sequence counts S and R, in the internal counters of the stations as well as in the transmitted frames, can be represented modulo 2M.

We note that similarly the counters L and A can use a representation modulo a constant. This constant depends on the size of the message buffer, and should at least be equal to 2M.

For proving these properties, it is useful to consider the following assertions which hold at any instant between the occurrences of the events (a), (b), or (c) defined above.

Assertions on the internal counter values (see for example figure 2):

(a) $N - M < S$ (b) $S \leq N$ (c) $N \leq L$
(d) $N - M \leq A$ (e) $A \leq \bar{R}$ (f) $\bar{R} \leq N$

Assertions on the content of the frames travelling over a line :

(g) $\bar{A} \leq R_n$ (h) $R_i \leq R_{i+1}$ for $n \leq i < \ell$ (i) $R_\ell \leq R$

(j) $S_i \leq N$ for $0 < i \leq \ell$

(k) m_i is the S_i-th message obtained from the source.

A corresponding set of assertions holds also for the opposite direction of transmission.

3.3 <u>Proof of the properties and assertions</u>

In this section, we use the protocol definition of section 3.1 and derive from it the properties and assertions stated in the section above. We first prove the assertions, and then the properties (1) through (4).

We prove the assertions and properties by induction on the number of events (a), (b), or (c) that have occurred since the initialization of the protocol. Clearly the assertions and properties hold for the initial state of the protocol. We now assume that they hold immediately before the occurrence of a single event (a), (b), or (c), and show in the following paragraphs that they then also hold immediately after the same event.

We first note that L is only changed by the event (a) of obtaining a new message, S and N by the event (b) of sending a frame, and R and A by the event (c) of receiving a frame. It is also clear from lines (2), (14), (9), and (18) that the values of L, A, N, and R can never decrease.

For assertion (a), one has to prove that it remains valid when the event (b) of sending a frame increases N or decreases S. When N is increased we have $S = N$, and since $M \geq 1$ we get $N - M < S$. The only way to decrease S is by assigning to it the value $A + 1$ (see line (7)). But then we have $A < S$ after the event, which together with assertion (d) yields assertion (a).

Assertion (b) follows from line (9). For assertions (c) and (d), one has to prove that they remain valid when the event (b) of sending a frame increases the value of N. Because of (9) this implies an increase of S, and $S = N$ holds after the event. The only way to increase S is by executing line (8). If, however, after the test of line (4) and (5), line (8) is executed, $S < L$ and $S < A + M$ must hold before the event, from which follow the assertions (c) and (d) afterwards.

Assertion (e) follows from the assertions (g), (h), and (i) for the opposite direction of transmission. Assertions (h) and (i) follow from the fact that R never decreases, and successive values of R are contained in the successive frames on the transmission line (see line 10). Assertion (g) follows from line (14) and assertion (h).

For proving assertion (f) we note that, because of lines (15) and (18), $\bar{R} = S_i$ where $\langle S_i, R_i, m_i \rangle$ is the last frame accepted by the receiving station, i.e. $i < n$. Assertion (f) then follows from assertion (j).

Assertion (j) follows from the fact that N is a non-decreasing function of time, that $S_\ell = S$ where S_ℓ is the sequence count of the last frame sent, and $S \leq N$ (assertion (b)).

Assertion (k) follows from the definition of the event (a) of obtaining a new message (lines (1), (2), and (3)) and line (10).

Property (1) is proved by supposing it holds immediately before the event (c) of receiving a frame. During this event, either no message is passed on to the sink (invalid redundancy check, reception out of order, or the sink is not ready) or the message passed on is the (R + 1)-th message obtained from the source (see assertion (k) and line (15)), and the counter R is incremented by one.

Property (2) follows immediately from property (1) and assertion (e), and property (3) follows from property (1) and $\bar{R} = L$ (the latter being a consequence of $A = L$ and assertions (e), (f), and (c)).

For proving property (4) we rewrite the lines (4) through (10) equivalently as :

```
if (A - S + M) = A - L + M
   or (A - S + M) = 0
then if  A = L  then  S:=  A
                else  S:=  A + 1
else  S:=  S + 1;
```

```
transmit-frame ( <S, R,
      send-buffer [ A - (A - S + M) + M ]> )
```

and the lines (14) and (15) as

$$A := A + (R_{frame} - A);$$

$$\text{if } (R - S_{frame} + M) = M - 1$$

We note that the value of N is not needed for the operation of the protocol, and that the case ">" can never occur in the comparisons of lines (4) and (5), as is shown by the assertions (b), (c), and (d). We shall show below that all the expressions in brackets (...) of the lines above have values that satisfy the relation $0 \leq \text{value} < 2M$. From this follows that an evaluation modulo 2M of these expressions does not introduce any ambiguities. Therefore we can introduce a modulo 2M representation for the sequence counts S and R without changing the operation of the protocol if, at the same time, we evaluate the expressions in brackets (...) modulo 2M.

We note that the values of S_{frame} and R_{frame} used in the event (c) of receiving a frame are exactly the values $\bar{S}_{\bar{n}}$ and $\bar{R}_{\bar{n}}$ of the frame $\langle \bar{S}_{\bar{n}}, \bar{R}_{\bar{n}}, \bar{m}_{\bar{n}} \rangle$ at the instance immediately before the event. The relations $0 \leq (A - S + M) < 2M$, $0 \leq (\bar{R}_{\bar{n}} - A) < 2M$, and $0 \leq (R - \bar{S}_{\bar{n}} + M)$ follow directly from the assertions. The remaining relation $(R - \bar{S}_{\bar{n}} + M) < 2M$ can be shown as follows: Be $\langle \bar{S}_i, \bar{R}_i, \bar{m}_i \rangle$ the last received and accepted frame, i.e. $\bar{S}_i = R$ and $i < \bar{n}$. We have $\bar{S}_i \leq \bar{N}^{(i)}$ from assertion (b) at the instance when the frame $\langle \bar{S}_i, \bar{R}_i, \bar{m}_i \rangle$ was sent by the opposite transmission station. Since N is a non-decreasing function of time we have $\bar{N}^{(i)} \leq \bar{N}^{(\bar{n})}$. Assertion (a) at the instant when the frame $\langle \bar{S}_{\bar{n}}, \bar{R}_{\bar{n}}, \bar{m}_{\bar{n}} \rangle$ was sent yields $\bar{N}^{(\bar{n})} < \bar{S}_{\bar{n}} + M$. These things together yield the relation $(R - \bar{S}_{\bar{n}} + M) < 2M$.

3.4 Remarks and extensions

We note that we have not dealt with the important problem of proving that a certain protocol state will actually be attained after a finite amount of time. For instance, we have proved the property (3) which states that if the protocol state is such that $A = L$ then all messages have been correctly delivered. But we have not shown that the protocol will reach such a state after a finite time interval. This question actually depends not only on the correct operation of the protocol, but also on the probability of transmission errors. For example, if the line brakes down completely, a state with $A = L$ will never be reached.

The problem of proving that a given state will actually be reached is related to the problem of proving that a given sequential program terminates. However, the operation of a communication protocol is non-deterministic due to the presence of errors. Therefore in general, such a termination proof would be difficult to obtain. In the case of finite state half duplex protocols, however, the approach of reference 4 can be used.

The retransmission scheme of the protocol discussed above is probably not the most efficient. It has been chosen for its simplicity. Several different schemes could be considered, such as retransmission of non-acknowledged frames after a fixed time interval ("time-out"), or selective retransmission with a more flexible receiving discipline. Window operation has been proposed for network access protocols [10]. These and similar transmission schemes can be described in the same manner as the simple protocol above, and corresponding assertions and properties can be proved for such protocols, using the same techniques.

4. Programming a Protocol

In this section we discuss some aspects of protocol implementation; we suppose in particular an implementation in software. As mentioned earlier, we believe that the implementation of a protocol is closely related to its design and logical verification. Writing a program that implements a given protocol on a given computer may introduce new errors (programming errors). Therefore, it would be advantageous to use a high-level implementation language so that only few transformations are necessary for obtaining the program from the original protocol definition. Using a high-level language and structured programming are likely to increase the reliability of the protocol implementation. We discuss in the following more specifically the description of parallel processes as they occur in protocols.

The concept of parallel processes is a useful structuring tool for the design of protocols. For example, the protocol introduced in section 2 may be understood as consisting, at each station, of the following processes: (1) a message producing source process, (2) a sending process, (3) a receiving process, and (4) a message consuming sink process. Programming tools for specifying inter-process communication and synchronization have been described in the literature [5,6]. The concept of monitors is particularly interesting. Monitors can be used to delimit the inter-process interaction and to specify explicitely the synchronization between the processes. They provide an efficient implementation tool [5,11].

The use of different levels of abstraction for the description of protocols is another useful design method. Often discussed in the literature on structured programming, this approach leads to the design of protocols in distinct levels [1,9]. For example, the protocol introduced in section 2 describes only a certain level of the whole communication system. The levels below concern, for instance, the redundancy check for error detection, or the modem line interface. The levels above may concern message routing in a communication network, or the exchange of messages for the communication between a resource and a "user".

Independent of these different levels in the design of a communication system, each level of the system may be described in more abstract or more detailed terms. For example, the description of the protocol in section 3.1 is relatively abstract. For an implementation of this protocol we would look for a relatively detailed specification, for instance in the form of a system written in some programming language for a given computer. In the remaining part of this section, we consider the protocol defined in section 3.1 as an example, and show how one can obtain for the same protocol successively more detailed descriptions, which lead to an implementation. We use in particular the concepts of parallel processes and monitors, and the notation of the programming language Concurrent Pascal [12].

As mentioned earlier, we consider four parallel processes on each side of the communication link. Figure 3 shows the system at one side of the link: There is a source process which generates messages, a sink process which consumes messages that come from the opposite side of the communication link, and the transmission station, which consists of one monitor which is called upon by the source and sink processes, and two processes that look after the transmission and reception of frames. The processes execute certain procedures which refer to the central monitor or the transmission lines, as indicated by the arrows. The sender and receiver processes are synchronized with the speed of the transmission line. Relative to this speed, the source and sink processes are synchronized by the control monitor.

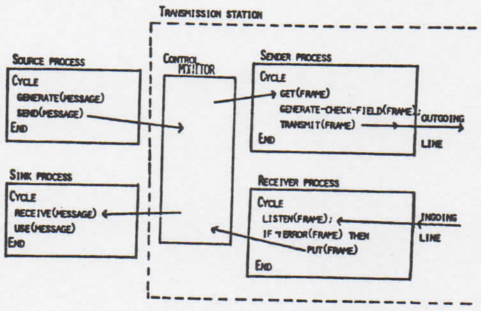

Figure 3: The structure of the communication system at one side of the transmission line.

The inner structure of the control monitor, not shown in Figure 3, is described by the Concurrent Pascal [12] program of the appendix. Essentially the monitor consists of the four procedures called upon by the processes. It enforces the mutual exclusion between the execution of these procedures. It contains the sequence counters L, A, S, and R, and a finite size *send-buffer*. Two queue variables are used to make the source or sink process wait, if necessary.

We note that very few logical changes have been introduced to the protocol definitions of the section 3.1 and 3.3 in order to obtain this program. We conclude that the language used is suitable for programming communication protocols. We hope that, in the future, sufficiently efficient implementations of languages for parallel processes and monitors will be available for the implementation of protocols.

We note that in Figure 3, we have assumed the existence of two additional processes, not shown in Figure 3 and may be implemented in hardware, which actually perform the continuous transmission and reception of bit strings on the two lines. We call these processes line processes. The procedures *transmit (frame)* and *listen (frame)* of the sender and receiver processes refer to the actions of these line processes. The line processes should not be interrupted by the sender or receiver processes when the latter execute a control monitor procedure. The interaction between the sender, receiver, and line processes could again be described by using monitors. However, we do not intend to give a description of the line processes in this paper.

5. Conclusions

We believe that the design, verification, and implementation of a protocol are inter-related activities. In this article, we have considered a particular protocol as an example, and have shown how certain tools, developped in different contexts, can be successfully applied to the logical verification and implementation of this protocol. We are confident that the same methods can also be applied in the case of the more complex protocols that are used in actual applications. The main objective of these efforts is to obtain more reliable and better documented communication systems.

We note that, apart from the tools discussed in this paper, other approaches (see for example in reference 3) have been described for dealing with the problems of logical verification and efficient implementation of communication protocols. Further research is needed for comparing the relative merits of these different methods.

There are at least two remaining problems which, in this paper, have been mentioned only briefly. The first problem (see section 3.4) is related to the logical verification of protocols, and is expressed in the following questions: What is the best method for proving that the operation of a given protocol, for a finite amount of information transfer, will terminate after a finite amount of time? What is the best method for showing that there are no undesired loops or dead-locks in the communication system?

The other problem is related to the implementation of protocols. We have proposed the use of a well-structured high-level language for describing parallel processes and monitors. This approach supposes that an efficient implementation of such a language exists on the computers that are used for the communication system. The author does not know of any language implementation of this kind that is available at present. However, several projects of implementation are in progress.

Acknowledgements:

I am grateful to Jean Vaucher for his interesting discussions, and many useful suggestions about the content of this paper.

APPENDIX: The control monitor of a transmission station written in Concurrent Pascal [12].

```
const
      mseq = 2 * M;
      mbuf = ...  {multiple of mseq} ;
type
      sequence-count = 0 .. mbuf -1;
      buffer-index = 0 .. mbuf -1;
      message-type = ...;
      frame-type = record S-frame, R-frame :
                                    sequence-count;
                          msg-frame: message-type;
                          check-field: ...
                   end;
var
control-monitor: monitor;
    var  send-buffer: array [ buffer-index ] of
                                    message-type;
         A,L: buffer-index;
         S,R: sequence-count;
         next-frame, buffer-free: queue;
         sink-waiting: boolean;
         sink-pointer: ↑message-type;
                       {pointer types represent an
                        extension to the language of
                        reference 12}
    procedure entry send (m: message-type);
         begin if (L-A) mod mbuf = mbuf -1
                          then wait (buffer-free) ;
              L: = (L + 1) mod mbuf;
              send-buffer [ L ] :=  m
         end;
    procedure entry receive (p: ↑message-type);
                          {p  is a pointer to where
                          the message is to be placed}
         begin  sink-waiting:=  true ;
                sink-pointer:=  p;
                wait (next-frame)
         end;
    procedure entry put (f: frame-type);
         var  increment: sequence-count;
         begin with  f  do begin
              increment:=  (R-frame - A) mod mseq;
              if increment > 0
              then begin
                   A:= (A + increment) mod mbuf;
                   signal (buffer-free)
                   end;
              if (R - S-frame + M) mod mseq = M - 1
                   and sink-waiting
              then begin
                   R:= (R + 1) mod mseq;
                   sink-pointer↑ := msg-frame;
                   sink-waiting:= false;
                   signal (next-frame)
                   end
              end
         end ;
    procedure entry get (f: frame-type);
         var  seq: sequence-count;
         begin     seq:= (A - S + M) mod mseq;
                   if seq= (A - L + M) mod mbuf
                      or seq = 0
                   then if A=L then S:= A mod mseq
                               else S:= (A + 1)mod mseq
                   else S:= (S + 1) mod mseq;
                   with f do begin
                        S-frame:= S;
                        R-frame:= R;
                        msg-frame:= send-buffer
                            [ (A + M - ^ ) mod mbuf ]
                        end        (A-S+M) mod mseq
         end ;
    begin  {monitor initialization}
         A:= L:= S:= R:= 0;
         sink-waiting:= false;
         clear (next-frame); clear (buffer-free)
    end;
```

REFERENCES:

[1] L. POUZIN, Network protocols, Nato International Advanced Study Institute, Brighton, Sept. 1973.

[2] C.A.R. HOARE, An axiomatic basis for computer programming, Comm. ACM, 12, p. 576 (1969).

[3] ACM Interprocess Communication Workshop, Santa Monica, Calif., March 1975.

[4] G.V. BOCHMANN, Communication protocols and error recovery procedures, in reference 3.

[5] P. BRINCH-HANSEN, Operating systems principles, Prentice Hall Inc., New Jersey, 1973.

[6] C.A.R. HOARE, Monitors: An operating system structuring concept, Communications ACM, 17, p. 549 (1974).

[7] ISO TC97/SC6, Document 1005.

[8] R.A. DONNAN and J.R. KERSEY, Synchronous data link control: a perspective, IBM Systems Journal, 13, p. 140 (1974).

[9] DATAPAC, Standard network access protocol, The computer communication group, Trans-Canada Telephone System, Nov. 1974.

[10] L. POUZIN, Basic elements of a network data link control procedure, ACM Computer Communication Review, Vol. 5, No. 1 (1975).

[11] A.R. SAXENA, An efficient implementation of monitors and condition variables, in reference 3.

[12] P. BRINCH-HANSEN, Concurrent Pascal - a programming language for operating systems design, Techn. Report No. 11, Information Science, Cal Tech, April 1974.