SPECIFICATION AND VERIFICATION OF AN HDLC PROTOCOL WITH ARM CONNECTION MANAGEMENT AND FULL-DUPLEX DATA TRANSFER*

A. Udaya Shankar and Simon S. Lam

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712

ABSTRACT

We use an event-driven process model to specify a version of the High-level Data Link Control (HDLC) protocol between two communicating protocol entities. The HDLC protocol is based upon the Asynchronous Response Mode (ARM) of operation, and uses the basic repertoire of HDLC commands and responses (with the exception of the CMDR response). It includes the features of poll/final cycles for connection management and checkpointing, sliding windows for data transfer, and ready/not ready messages for flow control. HDLC has three distinguishable functions: connection management, and one-way data transfers in opposite directions between the protocol entities. Various logical safety properties of the HDLC protocol concerning these functions have been verified using the method of projections.

1. INTRODUCTION

The High-Level Data Link Control (HDLC) protocol corresponds to a layer 2 protocol within the OSI reference model [ISO 79a, ISO 79b, ISO 80, ZIMM 80]. It is intended to provide reliable full-duplex data transfer between layer 3 protocol entities, using error-prone physical communication channels of layer 1. The specification of HDLC in the ISO documents defines precisely low-level protocol functions, such as error detection and frame synchronization. Formats of three types of frames specifying the encoding of control and data messages are also clearly defined. Aside from these basic definitions, however, the HDLC documents leave many options to be decided by the protocol implementor. In particular, one can choose from a variety of data link configurations and three operational modes that specify balanced or unequal relationships between the communicating entities. Also, various subsets of the messages can be used, instead of the entire set defined. Further, some aspects of HDLC are described informally in English and are not rigorously specified.

*This work was supported by National Science Foundation Grant No. ECS78-01803

In this paper, we use an event-driven process model [SHAN 82a] to specify a version of the HDLC protocol. (Refer to Figure 1.)

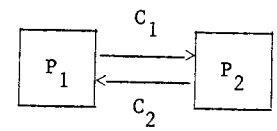

Figure 1. The protocol system model

Let P_1 denote the primary HDLC entity and P_2 the secondary HDLC entity operating in the Asynchronous Response Mode (ARM). C_1 and C_2 are (unreliable) communication channels. Our protocol uses the basic repertoire of HDLC commands and responses (with the exception of the CMDR response). It includes the use of poll/final cycles for checkpointing and connection management, timers for timeouts, sliding windows of size N for data transfers, and ready/not ready messages for flow control [ISO 79b]. Our protocol incorporates all of the principal HDLC functions.

HDLC has at least three distinguishable functions: connection management, and one-way data transfers in opposite directions. We state assertions that specify logical safety properties of the HDLC protocol concerning each function. These assertions have been verified to hold for the HDLC protocol specified herein [SHAN 82a, SHAN 82b].

1.1 The Method of Projections

A multi-function protocol such as HDLC is very complex and cannot be easily analyzed. Our analysis of the HDLC protocol has been achieved through an application of the method of projections [LAM 81, LAM 82a, LAM 82b, SHAN 82a] which breaks up the protocol analysis problem into smaller problems. The method of projections is described in detail in [LAM 82b, SHAN 82a]. Briefly, it constructs from a given multi-function protocol an image protocol for each of the functions that are of interest to us. An image protocol is specified just like any real protocol, and is obtained by retaining only those aspects of the multi-function protocol that are "relevant" to

the function being projected. Single-function image protocols are smaller than the original multi-function protocol and are thus easier to analyze. For example, the image protocol for HDLC connection management is similar to a handshake protocol [BOCH 78]. The image protocol for HDLC one-way data transfer is similar to other one-way data transfer protocols based on a sliding window mechanism [STEN 76], but augmented with initialization and checkpointing features.

An image protocol obtained by our construction procedure satisfies the following: any safety property that holds for the image protocol also holds for the original protocol. Additionally, if an image protocol satisfies a well-formed property then it is faithful. Informally, an image protocol is faithful if the following is true: any logical property, safety or liveness, concerning the projected function holds in the image protocol if and only if it also holds in the original protocol (see [LAM 82b, SHAN 82a] for a precise definition). The construction of well-formed image protocols involves an examination of protocol entities individually. There is no need to examine the global reachability space of the protocol interaction. Herein lies a significant advantage of the method of projections.

1.2 Summary of our Results

In Section 2 of this paper, we first describe an event-driven process model of a protocol system. Each component (entity or channel) of the protocol system is modeled as an event-driven process that manipulates a set of variables local to itself and interacts with adjacent components by message passing. The model includes several realistic protocol features such as as multi-field messages and the use of timers. This model is then used to specify the HDLC protocol.

In Section 3 of this paper, we state invariant safety assertions concerning the logical behavior of each of the functions. These assertions have been verified to hold for the HDLC protocol specified herein [SHAN 82a, SHAN 82b]. Due to lack of space, we have not included their proofs in this paper.

Proofs of the assertions as well as an exposition of the work presented in this paper can be found in [SHAN 82b]. Image protocols for the three HDLC functions are also presented there. In addition, inductively complete assertions stating the logical safety properties are shown and proved for each of the image protocols. (Assertions are inductively complete if (a) they are true at initialization of the protocol system, and (b) for each event in the protocol system, given that the assertions hold before the event occurrence, the specification of the event is sufficient to show that the assertions hold after the event occurrence.) From the properties of image protocols, it follows that these safety properties proved for the image protocols are also satisfied by the HDLC protocol.

Of the three image protocols presented in [SHAN 82b], only the image protocol for connection management is well-formed (hence faithful to the HDLC protocol for all safety and liveness properties concerning connection management), while the image protocols for the one-way data transfers are not well-formed (hence may not be faithful to the HDLC protocol for all safety and liveness properties concerning data transfer). In order for the data transfer image protocols to be well-formed, they have to be made substantially larger to account for dependencies in the HDLC protocol between the two one-way data transfer functions. For this reason, the HDLC protocol cannot be considered as well-structured. We then suggest a minor modification to HDLC that makes it well-structured, i.e., small well-formed image protocols can be constructed for each of its three functions.

The reader is referred to [LAM 82b, SHAN 82a] for a detailed treatment of the theory of projections and the method to construct image protocols.

2. AN HDLC/ARM PROTOCOL

In this section, we describe the HDLC/ARM protocol for two protocol entities. ARM denotes the Asynchronous Response Mode of operation. Let P_1 be the primary HDLC entity, and let P_2 be the secondary HDLC entity. P_1 sends messages to P_2 using channel C_1, and P_2 sends messages to P_1 using channel C_2 (see Figure 1). There is a user at entity P_1 and a user at entity P_2. The HDLC protocol system offers to the users a reliable connection that (a) can be opened/closed by the user at P_1, and (b) when open, allows each user to send data blocks to the other user in sequence (without loss, duplication or reordering). The HDLC protocol system offers three functions to the users: connection management, and one-way data transfers in two directions.

2.1 Assumptions about the Environment

To obtain assertions about the logical behavior of the protocol system, a few assumptions are needed about the environment in which HDLC operates. At any time, channel C_i contains a (possibly empty) sequence of messages sent by P_i, for i=1 and 2. Messages in the channels may be corrupted by noise, but not reordered or duplicated. When P_i sends a message, that message is appended to the tail of the message sequence in C_i. When the channel C_i is not empty, the first message (at the head of the message sequence) can be removed and passed on to P_j ($j \neq i$), provided that the message is not corrupted. If the message is corrupted, it is deleted and not passed on to P_j (we assume a perfect error-detection mechanism). The frame-level functions of HDLC [ISO 79a] such as the frame formatting of HDLC messages, bit insertion/deletion to make flags unique, error detection, etc., are not considered as part of the entities P_1 and P_2, but have been included in the channel model. Finally, messages in the channels have a bounded lifetime. The first message in channel C_i is deleted if it has been in the channel for a specified time, denoted by MaxDelay_i.

2.2 Event-driven Process Model

Each component of the protocol system (i.e., protocol entity or channel) is modeled as an event-driven process that manipulates a set of variables local to itself and interacts with adjacent components by message passing. An event-driven process consists of events. The events of an entity consist of message sends, message receptions and changes internal to the entity. The events of a channel correspond to transformations on the channel message sequence. An event can occur only if variables of the protocol system satisfy certain conditions, referred to as the enabling condition of the event. When an enabled event occurs, variables of the protocol system are affected. Whenever an event-driven process has enabled events, any one of them can occur. We assume fairness in the choice of the event to occur.

2.2.1 Time variables and time events

For HDLC to function correctly, it is neccessary that each HDLC protocol entity guarantees certain constraints on the time intervals between occurrences of events involving that entity. Also, recall that messages in channels have bounded lifetimes. Because (physical) time elapses at the same rate everywhere, these time constraints give rise to precedence relations between remote events in different components. Furthermore, these precedence relations are vital to the proper functioning of the HDLC protocol. We cannot adequately model such a time-dependent system by using only entity and channel events [SHAN 82c, SHAN 82a]. It is neccessary to relate the elapsed times measured at different components. We do this by introducing time variables in the components to measure elapsed time in integer ticks, and time events to age the time variables.

Each time variable takes its values from N_t = {Off,0,1,2,...}. A time variable is termed inactive if its value is Off, else it is termed active. The value of a time variable can be changed in only two ways. First, it can be aged by a time event. When an active time variable is aged, its value is incremented by 1; when an inactive time variable is aged, its value is not affected. Second, a time variable in a component can be reset to any value in N_t by a system event involving that component. Thus, for an active time variable, the difference between its current value and the value it was last reset to, indicates the time elapsed since the last reset.

We will use two types of time variables in our model: global time variables and local time variables. All global time variables in a system model are aged by the same time event, referred to as the global time event. Thus, all active global time variables are coupled. The global time event models the elapse of physical time in the protocol system model. Global time variables are typically used to model time constraints that are satisfied by components without the use of timers.

Local time variables are used to model the timers that are implemented in system components. To each local time variable t there is a unique local time event that ages t (and t alone). Thus, t is not directly coupled to any other time variable. To specify its accuracy, we associate with t a global time variable t* and a reset value t_0. Whenever t is reset, both t* and t_0 are reset to the same value. t* is affected by the global time event just like any other global time variable. The accuracy of local time variable t is specified by its accuracy axiom which bounds t-t* at any time. For example, the accuracy axiom $|t-t^*| \leq 1 + a(t^*-t_0)$ can specify a timer with maximum relative error a in its clock frequency (Off-Off is treated as 0).

In this model, neither the local time event of t nor the global time event can occur, if such an occurrence would violate the accuracy axiom. By placing additional constraints on the set of allowed values for time variables, other types of time constraints satisfied by a component can be modeled. For example, let t be a time variable that is reset to 0 by event e_1 and reset to Off by event e_2. Let D be a specified delay. Then, to model the time constraint that e_2 occurs no later than D time units since the occurrence of e_1, we include (t<D) in the enabling condition of the time event of t. Such constraints on time events are known as time axioms. (For a more detailed presentation, the reader is referred to [SHAN 82c, SHAN 82a].)

2.2.2 Messages of the protocol model

The messages of the protocol system have multiple fields, and are specified in terms of message types. A message type M is specified by a tuple of the form $(M,F_1,F_2,\ldots,F_n)$, where $n \geq 0$. The first component contains the name of the message type and is a constant. The other components (if any) are the fields of the message type. Each field is a parameter that can take values from a specified set. We shall refer to $(M,F_1,F_2,\ldots,F_n)$ as the format of message type M. The messages sent by each entity are specified by a list of such message types.

2.2.3 Variables of the entities and channels

Each protocol entity has a set of variables, each with a specified domain of values. Some of these variables can be auxiliary variables that are not implemented, but which are useful in the specification/verification of the protocol system. Also, some of these variables can be time variables used in modeling time constraints satisfied by the entity.

In channel C_i, we associate with every message in transit a global time value that indicates the time spent by that message in the channel. This time value is referred to as the age of the message. For channel C_i, we define $Channel_i$ as the variable that represents, at any time, the sequence of (message,age) pairs in C_i.

2.2.4 Events of the protocol model

The events of the protocol system model can be categorized into entity events, channel events, and

time events. We will describe them in that order.

There are three types of entity events. We describe these events for entity P_i.

1. For each message type M with format $(M,F_1,\ldots,F_n)$ sent by P_i, there is a Send_M event. This event is enabled if the values of the variables of P_i satisfy a specified enabling condition predicate. Its occurrence appends an M-type message $(M,f_1,\ldots,f_n)$ to the tail of $Channel_i$, and updates the values of variables of P_i (f_k is an allowed value of F_k).

2. For each message type M with format $(M,F_1,\ldots,F_n)$ sent by $P_j(j\neq i)$, there is a Rec_M event. This event is enabled if the entity variables of P_i satisfy a specified predicate, and the first message in $Channel_j$ is any M-type message $(M,f_1,\ldots,f_n)$. Its occurrence removes the message $(M,f_1,\ldots,f_n)$ from $Channel_j$, and updates the values of variables of P_i.

3. An internal event of P_i involves no messages. It is enabled if the entity variables of P_i satisfy a specified predicate. Its occurrence updates the values of the entity variables. Internal events are used to model interactions of the entity with its local user, channel controller, as well as timeouts and other internal transitions of the entity.

Note that both send and receive events affect the state of a channel, as well as the state of the entity.

We now describe the channel events. For i=1 and 2, the channel loss event for channel C_i is enabled whenever $Channel_i$ is not empty. Its occurrence deletes the first (message,age) pair in $Channel_i$. (Recall that the channel behavior in Section 2.1 assumes that only the first message in each channel may be lost.)

We now define the local time events and the global time event for the protocol model. For each local time variable t in P_i, there is a local time event whose occurrence ages t; this event is enabled if its occurrence does not cause t to violate its accuracy axiom or any time axiom involving t. There is one global time event whose occurrence ages all global time variables, including the age values in $Channel_1$ and $Channel_2$. This time event is enabled if its action does not cause any of the time or accuracy axioms to be violated, or result in an age value in $Channel_i$ that exceeds $MaxDelay_i$ for i=1 and 2.

For each entity, it is assumed that its implementation enforces mutual exclusion between the occurrences of events of that entity. Furthermore, we assume that simultaneous occurrences of events in different components of the protocol system can be represented as an arbitrary sequence of occurrences of the same events. This latter assumption is reasonable because events in communication protocol systems can usually be defined in such a way that their occurrences are instantaneous.

2.3 HDLC Messages

We shall now describe the HDLC messages that are sent by P_1 and P_2. Recall that messages are specified in terms of message types, and that the format of each message type is a tuple in which the first component is the name, and the other components are the fields. The interested reader should compare our message types with the three HDLC frame formats in [ISO 79a] and note the similarities.

Messages sent by P_1

We now list the message types sent by P_1. Each of these message types has a Poll_bit field (abbreviated as P field) that can take the value 0 or 1. Any message with the P field set to 1 is referred to as a Poll.

1. (U,P,Command) This U message type represents the Unnumbered frames sent by P_1 for connection management. The Command field can take the value SARM or DISC. SARM stands for Set Asynchronous Response Mode, and requests P_2 to go on-line. DISC stands for Disconnect, and requests P_2 to go off-line.

2. (I,P,Data,NS,NR) This I message type represents the Information frames sent by P_1 for transporting data blocks to P_2. Let DATABLOCKS denote the set of data blocks that can be transported by the HDLC protocol. The Data field contains a user data block, and can take any value from DATABLOCKS. NS and NR are sequence numbers that take values from {0,1,...,N-1}. (N is 8 for normal HDLC operation and 128 for extended HDLC operation.) NS is referred to as the send sequence number, and is used to identify the position of the data block in the sequence of user data blocks. Successive user data blocks are sent with increasing send sequence numbers (modulo N). NR is referred to as the receive sequence number, and indicates the send sequence number of the I frame from P_2 next expected at P_1. NR is an acknowledgement for data flowing in the reverse direction (i.e., from P_2 to P_1), and acknowledges all data blocks with send sequence numbers up to NR-1. Finally, an I frame with P field set to 1 indicates that P_1 is ready to receive data from P_2.

3. (S,P,RStatus,NR) This S message type represents the Supervisory frames sent by P_1 for flow control and acknowledgement. The RStatus field can take the value RR or RNR, indicating that P_1 is respectively Ready or Not Ready to receive data from P_2. The NR field is the receive sequence number and has been described above.

Messages sent by P_2

We now list the message types sent by P_2. Each of these message types has a Final-bit field (abbreviated as F field) that can take the value 0

or 1. Any message with the F field set to 1 is referred to as a Final. P_2 responds to a received Poll by sending a Final at the earliest opportunity.

1. (U,F,Response) This U message type represents the Unnumbered frames sent by P_2. The Response field can take the value UA or DM. UA stands for Unnumbered Acknowledgement, and is sent to acknowledge reception of, and compliance with a U command received from P_1. DM stands for Disconnected Mode, and is sent when P_2 is off-line as a response to any message (except for SARM) received from P_1.

2. (I,F,Data,NS,NR) This I message type represents Information frames sent by P_2. The Data, NS and NR fields are similar to those in the I frames sent by P_1 (except that the roles of P_1 and P_2 are interchanged). Also, an I frame with the F field set to 1 indicates that P_2 is ready to receive data from P_1.

3. (S,F,RStatus,NR) This S message type represents Supervisory frames sent by P_2. The RStatus and NR fields are similar to those in the S frames sent by P_1 (except that the roles of P_1 and P_2 are interchanged).

Note that message types sent by P_1 and P_2 have similar names. This should however cause no confusion. (The P and F fields actually occupy the same bit position in HDLC frames. That bit is referred to as the P/F bit [ISO 79a].)

2.4 Variables of the HDLC Protocol Entities

We now list the variables of the protocol entities.

Variables of P_1

P_1, the primary HDLC entity, has the following variables (the domain of each variable is also listed using a Pascal-like notation):

```
{The following variables are primarily used
 in the Poll/Final cycle}

Poll_bit          : (0,1);
Poll_Timer    : (Off,0,1,2,...,PollTimeoutValue);
                {local time variable}
$Poll_Timer       : (Off,0,1,2,....);
                    {global time variable
                     associated with Poll_Timer}
Poll_Retry_Count : (0,1,...,MaxRetryCount);

{The following variable is primarily used
 in connection management}
Mode              : (Open, Opening, Closed,
                     Closing, LinkFailure);

{The following variables are primarily used
 in sending data blocks to P2}
Source : array[0.. ] of DATABLOCKS;
         {history variable of data blocks}
  User_in, S, A, : 0.. ;  {pointers to Source}
  VS, VA, VCS : 0..N-1;
                {pointer variables modulo N}
  Checkpoint_Cycle : Boolean;
  Remote_RStatus : (RR,RNR);

{The following variables are primarily used
 in receiving data blocks from P2}

Sink : array[0.. ] of DATABLOCKS;
       {history variable of data blocks}
  User_out, R : 0.. ;  {pointers to Sink}
  VR : 0..N-1;  {pointer variable modulo N}
  Local_RStatus : (RR,RNR);
```

Variables of P_2

P_2, the secondary HDLC entity, has the following variables (along with their domains):

```
{The following variables are primarily used
 in the Poll/Final cycle}

Final_bit : (0,1);
$Response_Time : (Off,0,1,2,...,MaxResponseTime);
                 {auxiliary global time variable}

{The following variables are primarily used
 in connection management}

Mode       : (Open, Opening, Closed, Closing);
U_Response : (UA, DM, None);

{The following variables are primarily used
 in sending data blocks to P1}

Source : array[0.. ] of DATABLOCKS;
         {history variable of data blocks}
  User_in, S, A : 0.. ;  {pointers to Source}
  VS, VA, VCS : 0..N-1;
                {pointer variables modulo N}
  Checkpoint_Cycle : Boolean;
  Remote_RStatus : (RR,RNR);

{The following variables are primarily used in
 receiving data blocks from P1}

Sink : array[0.. ] of DATABLOCKS;
       {history variable of data blocks}
  User_out, R : 0.. ;   {pointers to sink}
  VR : 0..N-1;  {pointer variable modulo N}
  Local_RStatus : (RR,RNR);
```

(Note that many variables in P_1 and P_2 have the same names. Whenever this can cause ambiguity, we will qualify the variable names with 1 or 2; e.g., $Mode_1$, $Mode_2$.)

2.5 Events of the HDLC Protocol

The events of the HDLC protocol system are formally specified in Tables 1-4. An informal description follows in the succeeding subsections. The events of the entities are shown in Tables 1 and 2. The program statements in upper case (POLL_SENT, FINAL_RECEIVED, INITIALIZE_SEND_VARIABLES, etc.) stand for code segments that are shown in Table 3. When used in an entity event, the variables they refer to are the variables of that entity. We use the notation $\oplus$ and $\ominus$ to refer to addition modulo N and subtraction modulo N respectively. The time events of the HDLC protocol are specified in Table 4. The initial

state of this protocol is given by the following value assignments to the protocol system variables: Poll_bit=0, Poll_Timer=$Poll_Timer=Off, Poll_Retry_Count=0 and Mode=Closed in P_1; Final_bit=0, $Response_Time=Off, Mode=Closed and U_Response=None in P_2; both $Channel_1$ and $Channel_2$ are empty. We now describe the operation of the HDLC protocol informally.

2.5.1 Poll/Final cycle events

We first describe the P/F cycle involving the Poll and Final messages. Recall that P_2 responds to a received Poll by sending a Final at the earliest opportunity. A Poll is said to be outstanding (at P_1) if it has been sent and its acknowledging Final is being awaited. At any time, at most one Poll may be outstanding. Poll_bit set to 1 indicates that the next message sent by P_1 must be a Poll message. Final_bit set to 1 indicates that the next message sent by P_2 must be a Final message. Poll_Timer is used to measure the time elapsed since the last Poll was sent. When Poll_Timer is Off, there is no Poll that is outstanding and P_1 can send a Poll. Poll_Timer is started (reset to 0) when the Poll is sent. Poll_Timer is stopped (reset to Off) either when the acknowledging Final is received, or when a time duration PollTimeoutValue has elapsed. In the latter case, referred to as a Timeout event (see Table 1), P_1 presumes that either the Poll or the Final was lost.

Poll_Timer is treated as a local time variable, and $Poll_Timer is its associated global time variable. We shall consider its accuracy axiom to be $|Poll\_Timer - \$Poll\_Timer| \leq 1 + a(\$Poll\_Timer)$, where a is the maximum relative error in Poll_Timer's clock frequency. (Since any reset to Poll_Timer leaves it either Off or 0, there is no need to specify an associated reset value for Poll_Timer.)

$Response_Time is an auxiliary global time variable that is active if (and only if) a Poll has been received and its Final has not yet been sent; it then indicates the (global) time elapsed since the reception of the Poll. MaxResponseTime denotes the maximum time needed by P_2 to respond to a Poll. This time constraint is modeled by assuming that P_2 satisfies the following local time axiom: $Response_Time $\leq$ MaxResponseTime. By having PollTimeoutValue $>$ $(1+a)(MaxDelay_1 + PollResponseTime + MaxDelay_2)$, P_1 ensures that the following P/F cycle properties hold:

(i) A Final received at P_1 is the response to the last Poll sent by P_1.

(ii) A Poll received at P_2 was sent after the last Final sent by P_2 left channel C_2 (the Final may not have been received by P_1).

Poll_Retry_Count indicates the number of Timeouts that have occurred since the last Final was received. If this exceeds MaxRetryCount, P_1 assumes that the data link (either C_1, P_2 or C_2) has broken down, and enters a LinkFailure mode, which can be exited only by user intervention.

2.5.2 Connection management events

Mode in entity P_1 indicates the status of the data link as perceived by P_1. Open/Closed are stable states indicating that P_1 is on-line/off-line. The user sets Mode to Opening/Closing to request P_1 to open/close the data link with the remote user. P_1 then polls P_2 with appropriate U commands (SARM/DISC), and upon receiving acknowledgement sets Mode to Open/Closed. LinkFailure indicates that P_1 perceives the data link to have broken down (in our model, this is due to Poll_Retry_Count exceeding MaxRetryCount).

Mode in entity P_2 is similar to that in P_1, except that LinkFailure is not one of its allowed values. Open/Closed are stable states. Opening/Closing indicate that P_2 has received a SARM/DISC command and has not yet sent the UA acknowledgement. Once the acknowledgement is sent, Mode is set to Open/Closed. U_Response indicates the kind of U message to be sent by P_2.

2.5.3 Data transfer and flow control events

Next we describe the data transfer variables at P_1. Source is a history variable that records the data blocks given by the local user to P_1 to send to the remote user. User_in, S and A are three pointers to Source. User_in points to the location in Source into which the local user places his next data block. S points to the data block in Source to be next sent to P_1. A points to the data block in Source to be next acknowledged by P_2. (See Figure 2(a)). VS is referred to as the send state variable, and indicates the send sequence number of the next data block to be sent. VA is referred to as the acknowledgement state variable, and indicates the send sequence number of the data block to be next acknowledged. VS (VA) points to the same data block in Source as S (A). Checkpoint_Cycle and VCS are explained later. Remote_RStatus is RR (RNR) if the latest flow control information from P_2 indicates that P_2 is Ready (not Ready) to receive data. Note that data blocks Source[A], Source[A+1], ..., Source[User_in-1] have to be saved in a local send buffer of P_1. Let SBuffSize be the size of this buffer.

Sink is a history variable that records the data blocks received from P_2, and accepted for delivery to the local user. R and User_out are pointers to Sink. R points to the location in Sink in which to place the next data block received in sequence from P_2. User_out points to the data block in Sink to be next delivered to the local user. VR is referred to as the receive state variable, and indicates the sequence number of the data block next expected. (See Figure 2(b).) VR points to the same data block as R. Local_RStatus is RR (RNR) if P_1 is Ready (Not Ready) to receive data blocks from P_2. Note that data blocks Sink[User_out], Sink[User_out+1],...,Sink[R-1] have to be saved in a local receive buffer. Let RBuffSize denote the size of this buffer. Local_RStatus reflects whether this buffer is full or not.

The data transfer variables of P_2 are similar to those of P_1 (except that the roles of P_1 and P_2 are interchanged).

At each entity, data can be sent and received only when Mode is Open. At each entity, each time that Mode is set to Open, the data transfer variables are initialized as follows: User_in=S=A=0, VS=VA=VCS=0, Checkpoint_Cycle=False, Remote_RStatus=RR, User_out=R=0, VR=0, and Local_RStatus=RR.

We will now describe informally the data transfer from P_1 to P_2. (Let Figure 2(a) represent the Source in P_1, and Figure 2(b) represent the Sink in P_2.) When the user at P_1 wants to send a data block, he places it in Source[User_in], and increments User_in by 1. When P_1 sends an I frame, the Data field contains Source[S], the NS and NR fields contain the current values of VS and VR. S is incremented by 1 and VS by 1 (modulo N). When an I frame arrives at P_2, if its NS equals the current value of VR, and Local_RStatus equals RR, then P_2 accepts the data block in the data field and places it in Sink[R]. R is then incremented by 1, and VR by 1 (modulo N). When the user at P_2 extracts the data block from Sink[User_out], User_out is incremented by 1. When P_1 receives an NR, that NR points to some data block in Source[A], Source[A+1],...,Source[S]. VA is updated to equal NR, and A is updated to point to the data block now outstanding.

Because the sequence numbers are cyclic, the number of outstanding blocks must never exceed N-1 (i.e., S-A should always equal (VS-VA) mod N); otherwise, a received sequence number will not point to a unique outstanding data block. Whenever a Poll is sent when data blocks are outstanding, Checkpoint_Cycle is set to True, and VCS is set to (VS-1 mod N) the NS of the most recently sent data block. Checkpoint_Cycle is set to False either when an NR equalling or exceeding VCS is received, or when a Final is received. In the latter case, if the NR with the Final does not acknowledge VCS, P_1 concludes that I frames were lost (because of the P/F cycle properties). VS and VA are then set to equal the received NR. S and A are adjusted accordingly. This method of checking data transfer progress (and initiating retransmission if necessary) is referred to as checkpointing.

In addition, P_2 sends flow control information indicating its current Local_RStatus to P_1. This information is sent in S frames, as well as in I frames that have their P field set to 1.

The data transfer from P_2 to P_1, and its flow control is similar, except that the roles of the Polls and Finals are interchanged in checkpointing. However, note that P_1 can initiate a Checkpoint_Cycle whereas P_2 cannot.

2.5.4 Time events

The time events of the protocol system are shown in Table 4. Poll_Timer_Tick is the local time event for Poll_Timer. Global_Tick is the global time event of the system. The procedure Age (in the actions of the time events) ages all its argument time variables by one tick. Note that the global time event cannot age $Response_Time beyond MaxResponseTime, nor can it cause a message to stay in $Channel_i$ for longer than $MaxDelay_i$, nor can it cause Poll_Timer to be more inaccurate than as specified by its accuracy axiom. Similarly Poll_Timer_Tick cannot cause Poll_Timer to be more inaccurate than as specified by its accuracy axiom.

3. SAFETY PROPERTIES OF THE HDLC PROTOCOL

The HDLC protocol described has three distinguishable functions offered to its users: connection management, and one-way data transfers in opposite directions. We will now state assertions that specify logical safety behavior of the HDLC protocol concerning each of these functions. Our analysis of the HDLC protocol was done through the use of protocol projections. An image protocol was constructed from the HDLC protocol for each of the three functions of interest. These image protocols were then verified to satisfy the safety properties described below. From the properties of image protocols, these assertions also hold for the HDLC protocol [LAM 82b, SHAN 82a]. It was easier to obtain the assertions from the image protocols than from the HDLC protocol, because each image protocol is smaller, both in the number of variables and in the complexity of event descriptions. The image protocols and proofs are given in [SHAN 82a, SHAN 82b].

We note that the Poll/Final cycle displays a time-dependent behavior. The essence of this time-dependent behavior is captured by the following assertion:

Poll_Timer = Off => No Poll in $Channel_1$
and Final_bit = 0
and no Final in $Channel_2$

This says that when Poll_Timer is Off (e.g., immediately after a Poll_Timeout occurrence), sufficient time has elapsed since the last Poll was sent so that the following hold: (a) the Poll is no longer in $Channel_1$, (b) if the Poll was received by P_2, then the acknowledging Final has already been sent, and (c) the acknowledging Final is no longer in $Channel_2$. This assertion allows us to deduce the P/F cycle properties described in Section 2.5.1. The HDLC protocol has been verified to satisfy the above assertion [SHAN 82b]. (We note that our event-driven process model includes measures of time which have been incorporated expressly for verifying assertions of time-dependent behavior.)

We will now state the logical safety properties satisfied by the HDLC protocol concerning each of the three functions.

3.1 Safety Properties for Connection Management

For this HDLC protocol, the following assertions concerning connection management is invariant (proof in [SHAN 82b]):

1. $Mode_1$ = Open => $Mode_2$ = Open
and no U frame in $Channel_1$
and no U frame in $Channel_2$
and U_Response = None

2. $Mode_1$ = Closed => $Mode_2$ = Closed
and $Channel_1$ is empty
and $Channel_2$ is empty
and U_Response = None

These assertions specify that the offline/online states of P_1 and P_2 are synchronized, and that the channels are not utilized when the two entities are offline.

3.2 Safety Properties for P_1 to P_2 Data Transfer

For the function of one-way data transfer from P_1 to P_2, the following two desirable properties have been verified to hold for the HDLC protocol (proof in [SHAN 82b]):

If $Mode_1 = Mode_2$ = Open then
1. $Sink_2[i] = Source_1[i]$ for $0 \leq i < User\_out_2$
2. $0 \leq A_1 \leq S_1 < A_1 + N$

The first says that while the data link is open, data is transferred in sequence from P_1 to P_2. The second says that the maximum number of outstanding data blocks (hence the minimum storage requirement) at P_1 is N-1.

3.3 Safety Properties for P_2 to P_1 Data Transfer

For the function of one-way data transfer from P_2 to P_1 the following two desirable properties have been verified for the HDLC protocol (proof in [SHAN 82b]):

If $Mode_2 = Mode_1$ = Open then
1. $Sink_1[i] = Source_2[i]$ for $0 \leq i < User\_out_1$
2. $0 \leq A_2 \leq S_2 \leq A_2 + N$

The first says that while the data link is open, data is transferred in sequence from P_2 to P_1. The second says that the maximum number of outstanding data blocks (hence the minimum storage requirement) at P_2 is N-1.

4. CONCLUSION

We have used an event-driven process model to specify and verify a version of the HDLC protocol between two communicating protocol entities. The HDLC protocol specified is based upon the Asynchronous Response Mode (ARM) of operation, and includes all of its important features. It uses the basic repertoire of HDLC commands and responses (with the exception of the CMDR response), and includes the use of poll/final messages for checkpointing and connection management, timers for timeouts, sliding windows of size N for data transfers, and ready/not ready messages for flow control.

The HDLC protocol has three distinguishable functions: connection management, and one-way data transfers between the two protocol entities. We stated assertions that specify desired logical safety properties of the HDLC protocol concerning the three functions. These assertions have been verified to hold for the HDLC protocol specified. The verification was done through an application of the method of projections. The image protocols of HDLC and proofs of the assertions can be found in [SHAN 82b]. The theory of projections and the method to construct image protocols are presented in [LAM 82b, SHAN 82a].

REFERENCES

[BOCH 78] Bochmann, G. V., "Finite State Description of Communication Protocols," Computer Networks, Vol. 2, 1978.

[ISO 79a] International Standards Organization, "Data Communications--HDLC Procedures--Frame Structure," Ref. No. ISO 3309, 1979.

[ISO 79b] International Standards Organization, "Data Communications--HDLC Procedures--Elements of Procedures," Ref. No. ISO 4335, 1979.

[ISO 80] International Standards Organization, "Data Communications--HDLC Unbalanced Classes of Procedures," Ref. No. ISO 6159, 1980.

[LAM 81] Lam, S. S. and A. U. Shankar, "Protocol Projections: A Method for Analyzing Communication Protocols," Conf. Rec. Nat. Telecomm. Conf., November 1981, New Orleans.

[LAM 82a] Lam, S. S. and A. U. Shankar, "Verification of Communication Protocols via Protocol Projections," Proc INFOCOM'82, April 1982, Las Vegas.

[LAM 82b] Lam, S. S. and A. U. Shankar, "Protocol Verification via Projections," Tech. Rep. 207, Dept. of Computer Sciences, Univ. of Texas at Austin, August 1982.

[SHAN 82a] Shankar, A. U., "Analysis of Communication Protocols via Protocol Projections," PhD thesis, Dept. of Elec. Eng., Univ. of Texas at Austin, December 1982.

[SHAN 82b] Shankar, A. U. and S. S. Lam, "An HDLC Protocol Specification and its Verification using Image Protocols," Tech. Rep. 212, Dept. of Computer Sciences, Univ. of Texas at Austin, September 1982.

[SHAN 82c] Shankar, A. U. and S. S. Lam, "On Time-Dependent Communication Protocols and their Projections," Proc. 2nd Int. Workshop on Protocol Spec., Test. and Verif., May 1982, Idyllwild, CA.

[STEN 76] Stenning, N. V., "A Data Transfer Protocol," Computer Networks, September 1976.

[ZIMM 80] Zimmermann, H., "OSI Reference Model--The ISO Model of Architecture for Open Systems Interconnection," IEEE Trans. on Commun. COM-28(4), April 1980.

TABLE 1. EVENTS OF PRIMARY HDLC ENTITY P_1

Event Name	Enabling Condition	Action
1. User_req_conn	Mode ≠ Opening or Closing	Mode := Opening
2. User_req_disc	Mode = Open	Mode := Closing
3. User_puts_data	Mode = Open and (User_in-A<SbuffSize)	{User places data in Source[User_in]} User_in := User_in + 1
4. User_gets_data	Mode = Open and (R-User_out > 0)	{User extracts data block from Sink[User_out]} User_out := User_out + 1; if Local_RStatus = RNR then Local_RStatus := RR

5. Send_U	Mode = Opening
	or Mode = Closing
	and Poll_bit = 1
		if Mode = Opening then Command:=SARM;
		if Mode = Closing then Command:= DISC;
		put($Channel_1$, (U,1,Command));
		POLL_SENT

6. Rec_U	first($Channel_2$) = U	get($Channel_2$, (U,F,Response));
		if Response = DM then Mode := Closed;
		if (Response = UA) and (Mode = Closing)
		then Mode := Closed;
		if (Response = UA) and (Mode = Opening)
		then begin Mode := Open;
		INITIALIZE_SEND_VARIABLES;
		INITIALIZE_REC_VARIABLES
		end;
		if F = 1 then FINAL_RECEIVED

7. Poll_Timeout	Poll_Timer>PollTimeoutValue	Reset(Poll_Timer, Off); if Poll_Retry_Count < MaxRetryCount then Poll_Retry_Count := Poll_Retry_Count + 1 else Mode := LinkFailure
8. Request_Poll	Poll_Timer = Off	Poll_bit := 1
9. Send_I	Mode = Open and VS⊕VA < N-1 and S < User_in and Remote_RStatus = RR and not(Poll_bit = 1 and Local_RStatus = RNR)	put($Channel_1$, (I,Poll_bit,Source[S],VS,VR)); VS := VS ⊖ 1; S := S + 1; if Poll_bit = 1 then begin CHECKPOINT_SENT; POLL_SENT end
10. Send_S	Mode = Open	put($Channel_1$, (S,Poll_bit,Local_RStatus,VR)); if Poll_bit = 1 then begin CHECKPOINT_SENT; POLL_SENT end

11. Rec_I	Mode = Open
	and first($Channel_2$) = I
		get($Channel_2$, (I,F,Data,NS,NR));
		DATA_NS_RECEIVED; NR_RECEIVED;
		if F = 1 then begin CHECKPOINT_RECEIVED;
		FINAL_RECEIVED;
		Remote_RStatus := RR
		end

12. Rec_S	Mode = Open
	and first($Channel_2$) = S
		get($Channel_2$,(S,F,RStatus,NR));
		Remote_RStatus := RStatus; NR_RECEIVED;
		if F = 1 then begin CHECKPOINT_RECEIVED;
		FINAL_RECEIVED;
		end

TABLE 2. EVENTS OF SECONDARY HDLC ENTITY P_2

Event Name	Enabling Condition	Action
1. User_puts_data	Mode = Open and (User_in-A<SbuffSize)	{User places data block in Source[User_in]} User_in := User_in + 1
2. User_gets_data	Mode = Open and (R - User_out > 0)	{User extracts data block from Sink[User_out]} User_out := User_out + 1; if Local_RStatus = RNR then Local_RStatus := RR

3. Rec_U	first($Channel_1$) = U
	and U_Response ≠ UA
		get($Channel_1$, (U,P,Command));
		if command = SARM begin Mode := Opening;
		U_Response := UA
		end;
		if (Command = DISC) and (Mode = Open)
		then begin Mode := Closing;
		U_Response := UA
		end;
		if (Command = DISC) and (Mode = Closed)
		then U_Response := DM;
		if P = 1 then POLL_RECEIVED

4. Send_U	U_Response ≠ None	put($Channel_2$, (U,Final_bit,U_Response));
		U_Response := None;
		if Mode = Closing then Mode := Closed;
		if Mode = Opening
		then begin Mode := Open;
		INITIALIZE_SEND_VARIABLES;
		INITIALIZE_REC_VARIABLES
		end
		if Final_bit = 1 then FINAL_SENT

5. Send_I	Mode = Open and VS⊕VA < N-1 and S < User_in and Remote_RStatus = RR and not(Final_bit = 1 and Local_RSTATUS = RNR)	put($Channel_2$, (I,Final_bit,Source[S],VS,VR)); VS := VS ⊕ 1; S := S + 1; if Final_bit = 1 then begin CHECKPOINT_SENT; FINAL_SENT; end
6. Send_S	Mode = Open	put($Channel_2$, (S,Final_bit,Local_RStatus,VR)); if Final_bit = 1 then begin CHECKPOINT_SENT; FINAL_SENT end

7. Rec_I	first($Channel_1$) = I
	and U_Response ≠ UA
		get($Channel_1$, (I,P,Data,NS,NR));
		if Mode = Closed then U_Response := DM;
		if Mode = Open
		then begin
		DATA_NS_RECEIVED; NR_RECEIVED;
		if P = 1
		then begin CHECKPOINT_RECEIVED;
		POLL_RECEIVED;
		end
		end

8. Rec_S	first($Channel_1$) = S
	and U_Response ≠ UA
		get($Channel_1$, (S,P,RStatus, NR));
		if Mode = Closed then U_Response := DM;
		if Mode = Open then begin
		Remote_RStatus := RStatus;
		NR_RECEIVED;
		if P = 1
		then begin
		CHECKPOINT_RECEIVED;
		POLL_RECEIVED
		end
		end

TABLE 3. DETAILS OF CODE SEGMENTS USED IN TABLES 1-2

```
POLL_SENT::
    Reset(Poll_Timer, 0); Poll_bit := 0;

FINAL_RECEIVED::
    Reset(Poll_Timer, Off); Poll_Retry_Count := 0;

POLL_RECEIVED::
    Final_bit := 1; Reset($Response_Time, 0);

FINAL_SENT::
    Final_bit := 0; Reset($Response_Time, Off);

INITIALIZE_SEND_VARIABLES::
    User_in := 0; S := 0; A := 0; VS := 0; VA := 0;
    Checkpoint_Cycle := False;
    Remote_RStatus := RR;

INITIALIZE_REC_VARIABLES::
    User_out := 0; R := 0; VR := 0;
    Local_RStatus := RR;
```

```
DATA_NS_RECEIVED::
    if VR = VS and Local_RStatus = RR
       then begin Sink[R] := Data;
                  R := R + 1; VR := VR ⊖ 1;
                  if (R - User_out) = Rbuffsize
                     then Local_RStatus := RNR;
            end;

CHECKPOINT_SENT::
    if VS ≠ AS
       then begin Checkpoint_Cycle := True;
                  VCS := VS ⊕ 1
            end;

NR_RECEIVED::
    if Checkpoint_Cycle and NR ⊕ VA > VCS ⊕ VA
       then Checkpoint_Cycle := False;
    A := A + (NR ⊕ VA); VA := NR;

CHECKPOINT_RECEIVED::
    if Checkpoint_Cycle
       then begin Checkpoint_Cycle := False;
                  VS := VA; S := A
            end;
```

TABLE 4. TIME EVENTS FOR THE PROTOCOL SYSTEM

Event Name	Enabling Condition	Action
1. Poll_Timer_Tick	(Poll_Timer - $Poll_Timer) ≤ a($Poll_Timer)	Age(Poll_Timer)
2. Global_Tick	($Poll_Timer - Poll_Timer) ≤ a($Poll_Timer) and ($Response_Time < MaxResponseTime) and (all ages in $Channel_1$ < $MaxDelay_1$) and (all ages in $Channel_2$ < $MaxDelay_2$)	Age($Poll_Timer); Age($Response_Time); Age(all ages in $Channel_1$); Age(all ages in $Channel_2$);

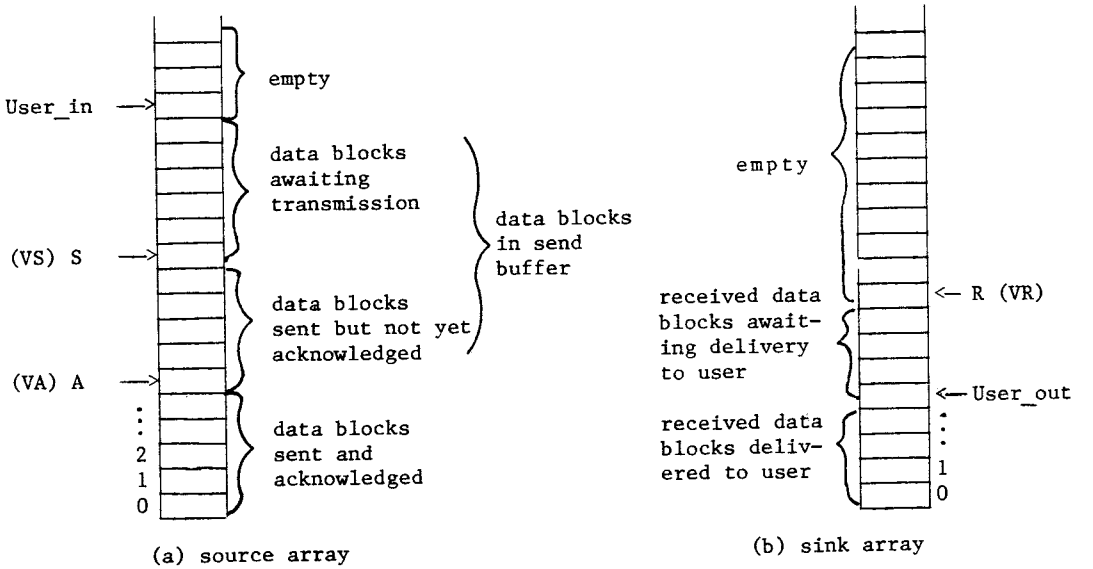

Figure 2. Pictorial representation of pointer positions for source and sink history arrays