An HDLC Protocol Specification and Its Verification Using Image Protocols

A. UDAYA SHANKAR and SIMON S. LAM
University of Texas at Austin

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 protocol is verified using the method of projections. The verification serves as a rigorous exercise to demonstrate the applicability of this method to the analysis of real-life communication protocols.

The HDLC protocol has two characteristics found in most real-life communication protocols. First, the HDLC protocol operates under real-time constraints that are important not only for its performance but also for its correct logical behavior. We specify this real-time behavior using time variables and time events. Second, the HDLC protocol has three distinguishable functions: connection management, and one-way data transfers between the protocol entities. For each of these functions, we construct an image protocol using the method of projections. With each image protocol we obtain inductively complete invariant assertions that state various desirable logical safety properties. From the properties of image protocols it follows that these safety properties as proved for the image protocols are also satisfied by the HDLC protocol presented herein. We also suggest a minor modification to HDLC that will make it well-structured.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol architecture, HDLC, protocol verification*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*real-time systems*; D.2.4 [**Software Engineering**]: Program Verification—*correctness proofs*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning About Programs—*assertions, invariants, pre- and post-conditions, specification techniques*

General Terms: Algorithms, Languages, Verification

Additional Key Words and Phrases: Communication protocols, data link control, HDLC protocol, method of projections, image protocols, message-passing networks, communicating processes

1. INTRODUCTION

The High-Level Data Link Control (HDLC) protocol corresponds to a layer 2 protocol within the OSI reference model [7, 8, 9, 23]. 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 precisely defines low-level protocol functions, such as error

This work was supported by NSF Grants ECS78-01803 and ECS83-04734.
This paper was submitted for publication in *Transactions on Computer Systems* by the authors and not by the SIGCOMM Symposium Program Committee. Author's addresses: A. U. Shankar, Department of Computer Science, University of Maryland, College Park, MD 20742; S. S. Lam, Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712.

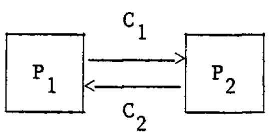

Fig. 1. The protocol system model.

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 among 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.

In this paper, we use an event-driven process model to specify a version of the HDLC protocol, and then apply the method of projections to verify it [13, 16, 20]. This verification serves as an exercise for demonstrating the applicability of this method (see Figure 1). P_1 is a primary HDLC entity and P_2 a 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, cyclic sequence numbers and sliding windows of size N for data transfers, and ready/not ready messages for flow control [8]. Our protocol incorporates all of the principal HDLC functions.

1.1 Analysis of Multifunction Protocols

The HDLC protocol has at least three distinguishable functions: connection management, and one-way data transfers in two directions. A multifunction protocol such as HDLC is very complex and cannot be easily analyzed. To reduce the complexity of analysis, an approach that appears attractive is to decompose each protocol entity into modules for handling the different functions of the protocol. For example, each protocol entity in HDLC may be decomposed into three functional modules as shown in Figure 2. Each module communicates with a corresponding module in the other protocol entity to accomplish one of the three functions. Bochmann and Chung [2] used a decomposition approach to specify a version of the HDLC protocol. However, the decomposition approach does not seem to facilitate analysis of the protocol. The main difficulty is that significant interaction exists among the modules. We identify two types of dependencies. First, modules interact through *shared variables* within an entity. Second, they also interact because data and control messages sent by different modules in one entity to their respective modules in the other entity are typically encoded in the same protocol message (*shared protocol messages*).

Most communication protocols that have been rigorously analyzed and presented in the literature are concerned with a single function: either a connection management function [1, 10, 14] or a one-way data transfer function [22, 6, 4]. For example, both safety and liveness properties have been formulated and proved for Stenning's protocol [22, 6]. Stenning's protocol is a one-way data transfer protocol. It corresponds to the interaction of a data send module and a

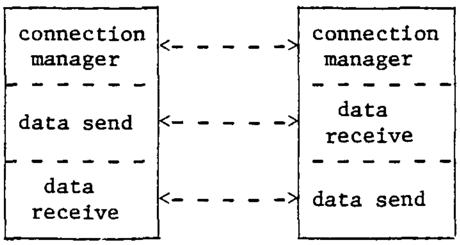

Fig. 2. Functional decomposition of HDLC.

data receive module in isolation (see Figure 2). Any interaction between these modules and other modules is not accounted for. As such, this protocol constitutes just one function of a real-life protocol such as HDLC. The following question arises: Are the safety and liveness properties that are proved for the one-way data transfer protocol still valid when it is implemented as part of a multifunction protocol with the two types of dependencies mentioned above?

We use the method of projections [11, 12, 13, 16, 20] to break up our HDLC protocol analysis problem into smaller problems. The theory of projections is described in [13] and [16]; we will not go into its details here. The projection method is different from the straightforward approach of decomposing protocol entities into functional modules. The objective is to construct from the given HDLC protocol an image protocol for each of the three functions that are of interest to us (referred to as the *projected functions*).

An *image protocol* is specified just like any real protocol. The states, messages and events of entities in an image protocol are obtained by treating groups of states, messages and events in the original protocol as equivalent and aggregating them. As a result, an image protocol is smaller than the original protocol. 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: Any logical property, safety or liveness, that can be stated for the image protocol holds in the image protocol *if and only if* it also holds in the original protocol. (*Fairness* in the scheduling of enabled events in the original protocol system is assumed; that is, no enabled event will be indefinitely delayed [15].)

The objective of our construction procedure is to generate the smallest image protocol that is of sufficient resolution to verify a desired logical property A_0 of the projected function. For example, the image protocol that is constructed for HDLC connection management is similar to a handshake protocol [1]. The image protocol for HDLC one-way data transfer is similar to other one-way data transfer protocols based on a sliding window mechanism, but is augmented with initialization and checkpointing features. There are two methods that we use to determine the desired resolution. The first method is applicable when A_0 is a safety (including real-time) property. The second method constructs a well-formed image protocol, and is applicable whether A_0 is a safety or liveness property. In each method, an initial resolution derived from A_0 is successively refined.

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 image protocol interaction or of the original protocol interaction. Given a multifunction protocol (such as HDLC), a well-formed image protocol can always be obtained for each function by increasing its resolution. However, the successful construction of well-formed image protocols that are much smaller than the original multifunction protocol depends upon whether the multifunction protocol has a good structure. Thus, one can think of a multifunction protocol as being *well-structured* if it possesses small well-formed image protocols for its functions.

1.2 Real-Time Behavior

Another important characteristic of real-life communication protocols is that they are *time-dependent* systems. Real-time constraints (such as bounded response times, packet lifetimes, etc.) exist within individual entities and channels [21]. These local time constraints give rise to global precedence relations between remote events. Such global relations are essential to the correct functioning of the protocol system. The modeling of real-time behavior has usually been neglected in previous protocol analyses.

To verify a time-dependent system, it is necessary to include measures of real time in the modeling of the protocol system. If time is not modeled explicitly, then one is forced to resort to informal arguments about global timing relations in the system. Such informal arguments are inadequate and are the source of many protocol system design errors [3].

The real-time behavior of communication protocols has implications regarding the formulation of liveness assertions. Typically, if a protocol does not achieve progress (transfer of data, establishment of a connection, etc.) within a bounded time duration T, then the protocol resorts to some alternative action (abort, reset, retransmission, etc.). The protocol will not wait for a finite but unbounded amount of time. Hence, a temporal logic liveness assertion [6] such as "eventually, a data block will be transferred" is not realistic. More appropriate is a real-time specification such as "if within a time duration T the data block is not transferred then at least n retransmissions of the data block have occurred, all of which failed, and the protocol has reset."

We model measures of time by incorporating time variables and time events into our protocol system model [17]. With time variables and time events, the real-time behavior of communication protocols can be stated by safety assertions; temporal logic liveness assertions are not needed.

1.3 Summary of This Paper

In Section 2, 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 multifield messages and the use of timers. This model is then used to specify the HDLC protocol.

In Section 3, we apply the method of projections to verify the HDLC protocol. In Section 3.1, we outline the definition of image protocols and two methods for obtaining image protocols. In Sections 3.2, 3.3 and 3.4, we construct from the given HDLC protocol an image protocol for each of the three functions that are of interest to us. For each image protocol, we obtain invariant safety assertions concerning some desired logical behavior of the projected function. From the properties of image protocols these assertions also hold for the entire HDLC protocol.

Of the three image protocols obtained, the image protocol for connection management is well-formed. However, the image protocols for the one-way data transfers are not well-formed. In order for these 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.

In Section 3.5, we describe a minor modification to the HDLC protocol that allows small well-formed image protocols to be constructed for each of the one-way data transfer functions, as well as for the connection management function. The invariant safety assertions obtained in Sections 3.2, 3.3 and 3.4 continue to hold for the connection management and data transfer image protocols of the modified HDLC protocol. The HDLC protocol with this modification can be regarded as a well-structured protocol.

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 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).

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 $i = 2$. 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 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 [7], such as the frame formatting of HDLC messages, bit insertion/deletion to make flags unique, error detecting, 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. 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.

2.2.1. *Time Variables and Time Events.* In HDLC, each 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. It is necessary 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 [16, 17].

Each time variable takes its values from $N_t = \{\text{Off}, 0, 1, 2, \ldots\}$. A time variable is termed *inactive* if its value is Off; otherwise 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 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 [16] and [17].)

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* 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. The messages sent by each entity are specified by a list of such message types. For simplicity, we often use M to refer to $(M, F_1, F_2, \ldots, F_n)$.

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 auxillary variables used only in 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 $(M, F_1, \ldots, F_n)$ 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 the variables of P_i(where f_k is an allowed value of F_k).
2. For each message type $(M, F_1, \ldots, F_n)$ sent by $P_j (j \neq i)$, there is a Rec_M event. This event is enabled if the 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 or 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 $i = 2$, 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 requires only that the first message in each channel may be lost. There is no loss of generality here because every message must become the first message in the channel before it can be received and checked for errors.)

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 $i = 2$.

For each entity, we assume mutual exclusion between the occurrence 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

Messages sent by P_1.

Each of the message types of P_1 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 $\{0, 1, \ldots, N - 1\}$. (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 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.

Each of the message types of P_2 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 [7].)

2.4 Variables of the HDLC 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_bit* = 1 indicates that the next message to be sent by P_1 is a Poll. Initially, *Poll_bit* = 0.}

Poll_Timer: (Off, 0, 1, 2, . . . , *PollTimeoutValue*);
   {*Poll_Timer* is a local time variable which is active ($\neq$ Off) if and only if a Poll is outstanding; an active *Poll_Timer* indicates the time elapsed since the Poll was sent. *Poll_Timer* is reset to Off either upon receiving the acknowledging Final, or when *Poll_Timer* = *PollTimeoutValue* (Timeout event). Initially, *Poll_Timer* = Off.}

$Poll_Timer: (Off, 0, 1, 2, . . .);
   {*$Poll_Timer* is the global time variable associated with *Poll_Timer*. Initially, *$Poll_Timer* = Off.}

Poll_Retry_Count: (0, 1, . . . , *MaxRetryCount*);
{*Poll_Retry_Count* indicates the number of Timeouts that have occurred since the last Final was received. Initially, *Poll_Retry_Count* = 0.}

{The following variable is primarily used in connection management.}

Mode: (Open, Opening, Closed, Closing, LinkFailure);
{*Mode* indicates the status of the data link as perceived by P_1. *Mode* is set to LinkFailure when *Poll_Retry_Count* exceeds *MaxRetryCount.* Initially, *Mode* = Closed.}

{The following variables are primarily used in sending data blocks to P_2.}

Source: array[0 . . ∞] of *DATABLOCKS;*
{*Source* is a history variable that records the data blocks given by the local user to P_1 to send to the remote user at P_2.}

User_in, S, A: 0 . . ∞;
{*User_in, S* and *A* are pointers to *Source* (see Figure 3a). *User_in* points to where the local user places his next data block. *S* points to the data block to be next sent to P_2. *A* points to the data block to be next acknowledged by P_2. All three pointers are intialized to 0 when the data link is opened (when *Mode* is set to the value Open). Data blocks from *Source*[*A*] to *Source* [*User_in* − 1] are saved in a buffer of size *SBuffSize.*}

VS, VA, VCS: 0 . . *N* − 1;
{*VS, VA* and *VCS* are pointers (modulo *N*) to *Source. VS*(*VA*) indicates the send sequence number of the next data block to be sent (acknowledged). *VS* and *VA* are initialized to 0 when the data link is opened. *VCS* is described below.}

Checkpoint_Cycle: Boolean;
{*Checkpoint_Cycle* is set to True when a Poll is sent and data is outstanding; *VCS* is set to the sequence number of the most recently sent data block. *Checkpoint_Cycle* is set to False when either the data block indicated by *VCS* is acknowledged, or a Final is received and that data block remains unacknowledged. In the latter case, retransmission of data blocks starting from *VCS* is initiated. *Checkpoint_Cycle* is initialized to False when the data link is opened.}

Remote_RStatus: (RR, RNR);
{*Remote_RStatus* indicates the data receive status of P_2. It is initialized to RR when the data link is opened.}

{The following variables are primarily used in receiving data blocks from P_2.}

Sink: array[0 . . ∞] of *DATABLOCKS;*
{*Sink* is a history variable that records the sequence of data blocks received from P_2 and to be delivered to the local user.}

User_out, R: 0 . . ∞;
{*User_out* and *R* are pointers to *Sink* (see Figure 3b). *User_out* points to the data block to be next delivered to the local user. *R* points to where the

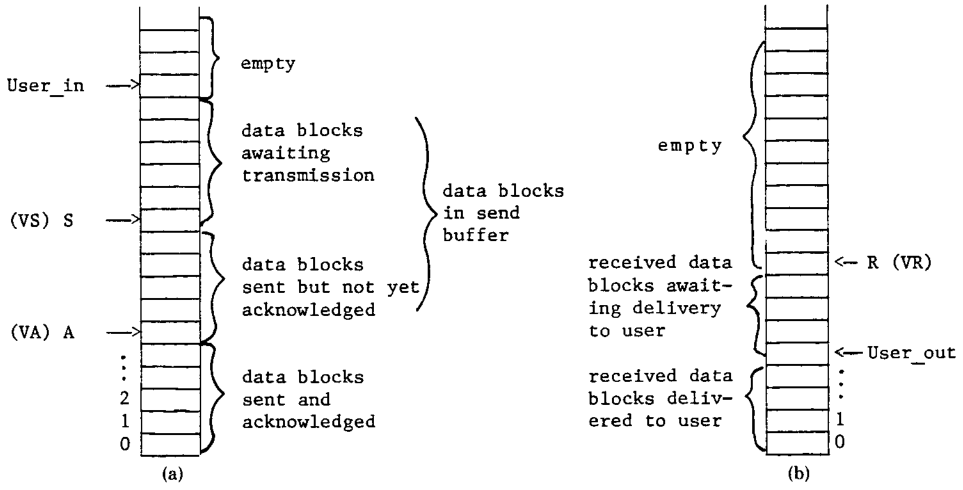
User_in
(VS) S
(VA) A
2
1
0
empty
data blocks awaiting transmission
data blocks sent but not yet acknowledged
data blocks sent and acknowledged
data blocks in send buffer
(a)
empty
received data blocks awaiting delivery to user
received data blocks delivered to user
R (VR)
User_out
1
0
(b)

Fig. 3. Pictorial representation of pointer positions for source and sink history arrays: (a) source array; (b) sink array.

next data block received in sequence from P_2 will be placed. Both pointers are initialized to 0 when the data link is opened. Data blocks from *Sink* [*User_out*] to *Sink*[*R* − 1] are saved in a buffer of size *RBuffSize.*}

VR: 0 .. $N - 1$;

{*VR* is a pointer (modulo *N*) to *Sink*, and indicates the sequence number of the data block next expected. *VR* is initialized to 0 when the data link is opened.}

Local_RStatus: (RR, RNR);

{*Local_RStatus* indicates the data receive status of P_1. It is initialized to RR when the data link is opened.}

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);

{*Final_bit* = 1 if and only if a Poll has been received and the acknowledging Final not yet sent. Initially, *Final_bit* = 0.}

$Response_Time: (Off, 0, 1, 2, . . . , *MaxResponseTime*);

{*$Response_Time* is an auxiliary global time variable which is active when *Final_bit* = 1; when active it indicates the time elapsed since receiving the Poll. Initially, *$Response_Time* = Off.}

{The following variables are primarily used in connection management.}

Mode: (Open, Opening, Closed, Closing);

{*Mode* indicates the status of the data link as perceived by P_2. Initially, *Mode* = Closed.}

U_Response: (UA, DM, None);
{*U_Response* indicates the kind of U message to be next sent by P_2. Initially, *U_Response* = None.}

{The meaning and usage of the remaining variables are as described for P_1, except that the roles of P_1 and P_2 are interchanged. Also, in checkpointing, the roles of Poll and Final are interchanged.}

{The following variables are primarily used in sending data blocks to P_1}.

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 P_1}.

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. Wherever this might cause ambiguity, we qualify the variable names with numerical subscripts, for instance, $Mode_1$ and $Mode_2$.

2.5 Events of the HDLC Protocol

The events of the HDLC protocol system are formally specified in Tables I–V (all tables may be found in Appendix B). (A prose description can be found in [19].) The events of the entities are shown in Tables I and II. The program statements in upper case (POLL_SENT, FINAL_RECEIVED, INITIALIZE_SEND_VARIABLES, etc.) stand for code segments that are shown in Table III. 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. *Poll_Timer* (in P_1) has the accuracy axiom $|Poll\_Timer - \$Poll\_Timer| \leq 1 + a(\$Poll\_Timer)$, where a is the maximum relative error in *Poll_Timer*'s clock frequency. P_2 satisfies the local time axiom $\$Response\_Time \leq MaxResponseTime$. Finally, in order to have at most one Poll outstanding at any time, we assume that $PollTimeoutValue > 1 + (1 + a)(MaxDelay_1 + MaxResponseTime + MaxDelay_2)$.

The channel events are specified in Table IV.

The time events of the HDLC protocol are specified in Table V. 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 time variable arguments 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.

The initial state of the HDLC protocol system is given by the following value assignments to the protocol system variables: both *Channel*$_1$ and *Channel*$_2$ are empty; entity variables have the initial values specified in Section 2.4. Note that some of the entity variables concerned primarily with data transfer functions are not initialized until the data link is opened (*Mode* set to the value of Open).

3. IMAGE PROTOCOLS AND SAFETY PROPERTIES

The HDLC protocol described offers three distinguishable functions to the users: connection management, and one-way data transfers in two directions. We would like to examine the logical behavior of the HDLC protocol with respect to these functions.

We note that these three functions are not independent of each other. Dependencies of the two types described in Section 1.1 (shared variables and shared protocol messages) are present. We mention some of them here. First, the Poll/Final cycle (a handshaking mechanism in the protocol) is used by all three functions; that is, Poll and Final messages, *Poll_bit*, *Poll_Timer*, *Poll_Retry_Count* and *Final_bit* are shared by all three functions. Second, the connection management function interacts with the data transfer functions at opening (when the data transfer variables are initialized), and at closing (when data transfer is inhibited); the variable *Mode* in each entity, and the SARM, DISC and UA messages are shared by all three functions. Third, the data transfers in the two directions interact through I frames that carry data in one direction and acknowledgment for data in the opposite direction. In addition, an incoming I frame with the Poll (Final) field set to 1 conveys flow control information for outgoing data.

Such dependencies present major obstacles for protocol analysis using a decomposition approach. We use the approach of protocol projections to obtain an image protocol for each function. In Section 3.1, we outline a procedure for constructing image protocols of sufficient resolution to verify desired logical properties of individual functions. We also briefly describe how to obtain inductively complete assertions that imply desired safety properties.

An assertion is *inductively complete* for a protocol system if (a) the initial condition of the protocol system satisfies the assertion, and (b) for each event in the protocol, given that the assertion holds before the event occurrence, the enabling condition and the action of the event are sufficient to show that the assertion holds after the event occurrence. In Sections 3.2, 3.3 and 3.4, we use the approach outlined in Section 3.1 to obtain image protocols for each of the HDLC functions, and state inductively complete assertions that imply some desired safety properties.

In Section 3.5, we describe a minor modification to HDLC that allows us to obtain small well-formed image protocols for the three functions of interest. The modification involves the addition of a one-bit flow control field to the HDLC information frames. These well-formed image protocols are faithful to the HDLC protocol. (We have not, however, investigated liveness properties of the HDLC protocol or of our modified version of the HDLC protocol in this paper.) We propose that this modified HDLC protocol can be considered to be well-structured.

3.1 Verification Via Projections

In this section, we briefly outline the method of protocol projections. In Section 3.1.1, we define image protocols. In Section 3.1.2, we describe two iterative methods for generating an image protocol of sufficient resolution to verify a specified property A_0 of a given function. For a detailed and formal presentation of the theory and methodology, see [13, 16, 20].

3.1.1. *Image Protocol Definition.* An image protocol is defined with respect to a given subset of entity variables of the original protocol. For $i = 1$ and $i = 2$, let V_i denote the set of entity variables of P_i in the HDLC protocol, and let V_i' denote the subset of V_i representing the entity variables of an image protocol. Messages and events of the image protocol are defined based on V_1' and V_2'.

Each value assignment to the variables in V_i represents a state s of entity P_i. The portion of this value assignment that corresponds to variables in V_i' represents the image s' of s.

For each message type $(M, \underline{F})$ sent by P_i (where $\underline{F}$ denotes the fields of M), its image is denoted by $(M', \underline{F}')$, where $\underline{F}'$ is obtained by deleting those fields in $\underline{F}$ that do not affect variables in V_j' $(j \neq i)$ in the Rec_M event of P_j. The image of any M-type message $(M, \underline{f})$ is given by $(M', \underline{f}')$, where $\underline{f}'$ consists of those field values in $\underline{f}$ corresponding to $\underline{F}'$.

Each HDLC entity event specifies a set of entity state transitions (which involve messages in the case of send and receive events). For example, the Send_M event, which involves variables in V_i and fields in $\underline{F}$, specifies transitions of the form $(s, r, (M, \underline{f}))$; that is, when P_i is in some state s satisfying the enabling condition of Send_M, the action of Send_M leaves P_i in some state r and sends message $(M, \underline{f})$. The image of Send_M is an event Send_M' involving V_i' and $\underline{F}'$, and satisfying the following: Send_M' specifies a transition $(s', r', (M', \underline{f}'))$ if and only if Send_M specifies a transition $(s, r, (M, \underline{f}))$ for some s, r and $(M, \underline{f})$ whose images are s', r' and $(M', \underline{f}')$ respectively. The images of receive events and internal events are similarly defined. (The image events were obtained directly from the HDLC event descriptions without explicitly considering their state transitions [16].)

The image message types and the image entity events serve as the message types and entity events of the image protocol. Also, image message types and image entity events that have no effect on variables in V_1' and V_2' are eliminated in the image protocol; those that have identical effects are merged. The behavior of the communication channels is the same in the image protocol system as in the HDLC protocol system. The initial values of the image protocol variables are the same as the initial values of the corresponding HDLC variables.

An image protocol as constructed above captures only part of the behavior of the original protocol. However, we have shown that a safety property that holds in the image protocol also holds in the original protocol. Also, well-formed image protocols are faithful to the original protocol in all of their safety and liveness properties. Informally, an image protocol is well-formed if the following holds: in the image protocol, if an entity event e' takes the entity from state r' to state s' and involves message m', then in the original protocol, from every entity state r whose image is r' there is a sequence of entity events that will take the original protocol to a state where an event e whose image is e' can occur [13, 16].

We have not investigated liveness properties of the HDLC protocol. We can decide whether certain states will be eventually reached (a typical liveness assertion) by examining the execution paths of the system. The expressive power of our model for such liveness assertions is limited, when compared with models that incorporate temporal logic semantics [15]. Recall however that communication protocols are time-dependent systems, and that progress properties of such systems can be stated as safety assertions involving time variables.

3.1.2. *Constructing an Image Protocol of Desired Resolution.* We next describe an iterative method that attempts to find the smallest image protocol that is of sufficient resolution to verify a desired safety property A_0 of the projected function.

For $i = 1$ and $i = 2$, let V_i' denote the entity variables of P_i that appear in the assertion A_0. We first describe the iterative step. Construct an image protocol using V_1' and V_2' as the set of entity variables. Verify if assertion A_0 holds in this image protocol (see below). If it does, then from the properties of image protocols A_0 holds in the HDLC protocol and the verification is over. If A_0 does not hold in the image protocol, then there is a sequence of events, referred to as a *test sequence*, that takes the image protocol from its initial condition to global state that violates A_0 (see below). Consider the event sequences in the HDLC protocol that have images equal to the test sequence. If any of these HDLC event sequences can occur in the HDLC protocol, then A_0 does not hold in the HDLC protocol, and the verification is over. If none of these HDLC event sequences can occur, that is because some HDLC variables (not included in $V_1' \cup V_2'$) inhibit certain events from occurring. Include these variables in V_i' and repeat the above iterative step until termination. In the worst case, termination occurs with the image protocol being equal to the original protocol.

Although this approach may at first appear to be inefficient, since in each iteration we check whether A_0 holds for an image protocol, this is in fact not the case because at each iteration, even if we cannot verify A_0, we can usually establish properties of the image protocol that are helpful in verifying A_0. (For example, the image protocol may not transfer data correctly, but it does ensure correct Poll/Final handshake and data link initialization.) Since each succeeding image protocol is a refinement of all earlier image protocols, these properties, once established, need not be verified again.

To verify A_0 for an image protocol, we do an iterative search for an inductively complete safety assertion A that implies A_0. Initially A equals A_0. The iterative step is as follows. For each event e of the image protocol, determine the weakest precondition [5] that must hold before the occurrence of e in order that A holds immediately after the occurrence of e. Let C denote the conjunction of A and all the weakest preconditions. If C is equivalent to A and the initial state of the image protocol satisfies C, then A is inductively complete, A_0 holds for the image protocol, and the search is over. If the initial state of the image protocol does not satisfy C, then by examining the trace of the iterations, a test sequence can be determined that takes the image protocol from its initial state to a state violating A_0. In this case, A_0 does not hold and the search is over.

If neither of the above conditions holds, then replace A by C and repeat the iteration. If at any point we determine (by inspection) that a safety property T holds for the image protocol, then A can be replaced by the conjunction of A and T. This speeds up the search by constraining the weakest preconditions.

The above iterative method can be used only for verifying safety assertions. A variation on this method can be used when A_0 is either a safety or liveness assertion. In this second method, each image protocol that is obtained is tested for well-formedness (instead of testing whether A_0 holds). If the test for well-formedness fails, then the failure will point out additional HDLC entity variables to include in constructing the next image protocol. Repeat this step until a well-formed image protocol is obtained. Since well-formed image protocols are faithful, any outcome which results from verifying A_0 on the image protocol is valid for the original protocol.

Note that the check for well-formedness involves an examination of each protocol entity individually. It does not involve an analysis of the global interaction of the intermediate image protocols. Hence, this checking can be performed efficiently. On the other hand, given a safety assertion A_0, the image protocol resulting from the first method is usually smaller (never larger) than that obtained from the second method. Since real-life communication protocols are typically time-dependent systems with real-time specifications stated as safety properties, the first method is usually more useful in practice.

3.2 Image Protocol for Connection Management

The first image protocol we show is for the connection management function. A desirable property of the HDLC protocol with respect to this function may be stated as follows:

$$(Mode_1 = \text{Open} \Rightarrow Mode_2 = \text{Open}) \text{ and } (Mode_1 = \text{Closed} \Rightarrow Mode_2 = \text{Closed})$$

This property can be stated using only the variables *Mode* in P_1 and *Mode* in P_2. Starting from this initial set of entity variables, and applying the first method, we determine that the following variables are also needed: *Poll_bit, Poll_Timer, $Poll_Timer* and *Poll_Retry_Count* in P_1; *Final_bit, $Response_Time* and *U_Response* in P_2. The resulting image protocol (described below) is also well-formed—using the second method would result in the same image protocol.

The images of the HDLC message types can now be defined as follows. First, there are the message types sent by P_1. The image of message type (U, *P*, *Command*) is defined by (U′, *P*, *Command*) (i.e., all the fields of the U Message type are needed in the image protocol). The image of both (I, *P*, *Data*, *NS*, *NR*) and (S, *P*, *RStatus*, *NR*) can be defined by (I′ *P*) where I′ denotes a (new) message type that corresponds to either an I or an S frame of the HDLC protocol. Next, there are the message types sent by P_2. The image of message type (U, *F*, *Response*) is (U′, *F*, *Response*). The image of both (I, *F*, *Data*, *NS*, *NR*) and (S, *F*, *RStatus*, *NR*) can be defined by (I′, *F*). Thus (U′, *P*, *Command*) and (I′, *P*) are the message types sent by P_1, and (U′, *F*, *response*) and (I′, *F*) are the message types sent by P_2.

The entity events of the image protocol are displayed in Tables VI and VII. The channel and time events of the image protocol are exactly as the channel and time events shown in Tables IV and V for the HDLC protocol. These events are obtained by taking the images of the corresponding events of the HDLC protocol (Tables I, II, IV and V). In the event Send_U′ of Table VI, the enabling condition and action are the same as in the event Send_U of Table I, except that U is replaced by U′. This notation is also used in the remaining tables.

The initial state of this image protocol is obtained from the HDLC protocol: $Mode_1 = Mode_2 =$ Closed, $U\_Response =$ None, $Poll\_Timer = \$Poll\_Timer = \$Response\_Time =$ Off, $Poll\_bit = Final\_bit = 0$, $Poll\_Retry\_Count = 0$, and $Channel_1$ and $Channel_2$ are empty.

The reader is referred to [16] for additional details.

3.2.1. *Safety properties.* For this image protocol, the following assertion concerning the Poll/Final cycle has been shown to be inductively complete, hence invariant. (A proof can be found in Appendix A. It is shown as an illustration of our technique. Proofs of other assertions to be introduced below are omitted for brevity.) In the assertions, $\$Age$ denotes the age of the associated message in the channel.

Poll/Final (PF) Assertions.

PF1. $Poll\_bit = 1 \Rightarrow Poll\_Timer =$ Off

PF2. $Poll\_Timer =$ Off $\Rightarrow$ No Poll in $Channel_1$
and $Final\_bit = 0$
and no Final in $Channel_2$

PF3. (Poll, $\$Age$) in $Channel_1 \Rightarrow Poll\_Timer \neq$ Off
and $\$Poll\_Timer = \$Age$
and exactly one Poll in $Channel_1$
and $Final\_bit = 0$
and no Final in $Channel_2$

PF4. $Final\_bit = 1 \Rightarrow Poll\_Timer \neq$ Off
and $\$Poll\_Timer \leq MaxDelay_1 + \$Response\_Time$
and no Poll in $Channel_1$
and no Final in $Channel_2$

PF5. (Final, $\$Age$) in $Channel_2 \Rightarrow Poll\_Timer \neq$ Off
and $\$Poll\_Timer \leq MaxDelay_1 + MaxResponseTime + \$Age$
and no Poll in $Channel_1$
and $Final\_bit = 0$
and exactly one Final in $Channel_2$

For this image protocol, the conjunction of the PF assertions and the following assertions concerning connection management are inductively complete, hence invariant (proof in [16, 18]).

Connection Management (CM) Assertions.

CM1. (a) $Mode_1 =$ Open $\Rightarrow Mode_2 =$ Open and no U′ frames in $Channel_1$
and no U′ frames in $Channel_2$ and $U\_Response =$ None

(b) $Mode_1$ = Closed ⇒ $Mode_2$ = Closed and $Channel_1$ is empty
and Channel$_2$ is empty and *U_Response* = None
(c) (I′, *P*) in $Channel_1$ ⇒ $Mode_2$ = Open

CM2. *Poll_Timer* = Off ⇒ no U′ frames in Channel$_1$
and no U′ frames in $Channel_2$ and *U_Response* = None
and ($Mode_2$ = Open or $Mode_2$ = Closed)

CM3. (U′, *P*, ***Command***) in $Channel_1$ ⇒ exactly one U′ frame in $Channel_1$
and (U′, 1, ***Command***) is at tail of $Channel_1$
and (***Command*** = SARM ⇒ $Mode_1$ = Opening)
and (***Command*** = DISC ⇒ $Mode_1$ = Closing)
and ($Mode_2$ = Open or $Mode_2$ = Closed) and *U_Response* = None
and no U′ frames in $Channel_2$

CM4. *U_Response* ≠ None ⇒ *Final_bit* = 1
and $Channel_1$ is empty and $Channel_2$ has no U′ frames
and (*U_Response* = UA ⇒ ($Mode_1$ = $Mode_2$ = Opening
or $Mode_1$ = $Mode_2$ = Closing))
and (*U_Response* = DM ⇒ ($Mode_1$ = Closing
and $Mode_2$ = Closed))

CM5. (U′, *F*, ***Response***) in $Channel_2$ ⇒ $Channel_1$ empty
and exactly one U′ frame in $Channel_2$ and *F* = 1
and (***Response*** = DM ⇒ $Mode_2$ = Closed and $Mode_1$ = Closing)
and (***Response*** = UA ⇒ ($Mode_2$ = Closed and $Mode_1$ = Closing)
or ($Mode_2$ = Open and $Mode_1$ = Opening))
and *U_Response* = None

3.3 Image Protocol for P_1 to P_2 Data Transfer

We now consider the function of one-way data transfer from P_1 to P_2. Two desirable properties of the HDLC protocol with respect to this function may be stated as follows.

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

The first property states that data is transferred in sequence; the second that the maximum number of outstanding data blocks (and therefore the minimum storage requirement) at P_1 is $N - 1$. These properties can be stated using only the variables *Mode*, *Source*, *A* and *S* in P_1, and *Mode*, *Sink* and *User_out* at P_2.

Starting from this initial set of entity variables, and applying the first method, we determine that the following variables are needed: *Poll_bit*, *Poll_Timer*, *$Poll_Timer*, *Poll_Retry_Count*, *User_in*, *VS*, *VA*, *VCS*, *Checkpoint_Cycle* and *Remote_RStatus* in P_1, and *Final_bit*, *$Response_Time*, *U_Response*, *R*, *VR* and *Local_RStatus* in P_2. The resulting image protocol (described below) has sufficient resolution to verify the desired safety property. However, it is not well-formed. In fact, we have shown in [16] that to obtain a well-formed image protocol for this function, we would have to include almost the entire HDLC protocol.

We now define the images of the HDLC message types sent by P_1. The image of message type (U, *P*, *Command*) is (U′, *P*, *Command*). The image of (I, *P*, *Data*, *NS*, *NR*) is defined as (I′, *P*, *Data*, *NS*), where I′ is a new message type that corresponds to an I frame of the HDLC protocol. The image of (S, *P*, *RStatus*, *NR*) is (S′, *P*), where S′ is a new message type corresponding to an S frame of the HDLC protocol.

Next, we define the images of the HDLC message types sent by P_2. The image of (U, *F*, *Response*) is (U′, *F*, *Response*) . The image of (I, *F*, *Data*, *NS*, *NR*) is (I′, *F*, *NR*), where I′ denotes a new message type corresponding to an I frame of the HDLC protocol. The image of (S, *F*, *RStatus*, *NR*) is (S′, *F*, *RStatus*, *NR*) .

Thus (U′, *P*, *Command*), (I′, *P*, *Data*, *NS*) and (S′, *P*) are the message types sent by P_1, and (U′, *F*, *Response*), (I′, *F*, *NR*) and (S′, *F*, *RStatus*, *NR*) are the message types sent by P_2.

The events of the image protocol system are obtained by taking the images of the HDLC protocol system [16]. The events of the protocol entities in the image protocol are shown in Tables VIII and IX. The channel and time events for this protocol are exactly as described in Tables IV and V. The initial state of this protocol is the same as that of the HDLC protocol.

We note that the image protocol does in fact display a minor liveness property that the original protocol does not have: the number of I′ frames in channel C_2 can exceed *N*. In fact, the Send_I′ event of P_2 is the only event that causes the image protocol not to be well-formed [16]. In Section 3.5, we suggest a minor modification to HDLC that will correct this and allow us to construct small well-formed image protocols for the one-way data transfer functions, as well as for the connection management function.

3.3.1. *Safety Properties.* For this image protocol, we have verified (by using the method described in Section 3.1.2) the desired safety properties of the projected function.

Notation.

We first describe some notation that is used in the assertion. Recall that $Channel_1$ and $Channel_2$ represent the sequence of messages in C_1 and C_2 respectively. We shall think of $Channel_1$ ($Channel_2$) as a sequence of messages from left to right; the tail of $Channel_1$ ($Channel_2$) is at the left, and the head of $Channel_1$ ($Channel_2$) is at the right. When $Channel_1$ ($Channel_2$) contains only one message, the tail and the head point to the same message.

The following notation is used in describing the state of $Channel_1$. Given integers s and vs such that $s \geq 0$ and $0 \leq vs < N$, the notation $\langle s, vs \rangle$ denotes the tuple (I′, *Data*, *NS*), where *Data* contains $Source[s]$ and *NS* contains vs. Let $\underset{\sim}{x}$ be a sequence whose elements are either U′ frames or entries of the form $\langle s, vs \rangle$. $Channel_1$ is said to *satisfy* $\underset{\sim}{x}$, if, by deleting all S′ frames in $Channel_1$ and by deleting the *P* field in all I′ frames in $Channel_1$, the resulting sequence equals $\underset{\sim}{x}$. Given two consecutive elements x_1, x_2 in $\underset{\sim}{x}$, we say that a Poll is *to the immediate left* of x_2 to mean that it is in between x_2 and x_1. We say that a Poll is *to the left* of x_2 to mean that it is anywhere to the left of x_2.

Given integers n, s and vs such that $0 \leq n \leq s$ and $0 \leq vs < N$, the notation $\langle s - 1, vs \ominus 1 \rangle .. \langle s - n, vs \ominus n \rangle$ denotes the sequence $\langle s - 1, vs \ominus 1 \rangle$, $\langle s - 2,$

$vs \ominus 2\rangle, \ldots, \langle s - n, vs \ominus n\rangle$ if $n > 0$, and the empty sequence if $n = 0$. Either way, n is the length of the sequence. (We use a simple comma to denote concatenation.)

Finally, whenever the term [*Old_Info_Sequence*] appears in the assertions, it denotes any sequence (possibly empty) of entries of the form $\langle s, vs\rangle$. We use [*Old_Info_Sequence*] to refer to the $\langle s, vs\rangle$ sequence obtained from the I′ frames in $Channel_1$ pertaining to a data connection that is in the process of being reset.

The following notation is used in describing the state of $Channel_2$. Given an integer vr such that $0 \le vr < N$, the notation $[vr]$ denotes either an empty sequence or any sequence of *one* or more receive sequence numbers, each equalling vr. Let $\underset{\sim}{y}$ be a sequence whose elements are either U′ frames or entries of the form $[vr]$. An instance of $\underset{\sim}{y}$ is any sequence of U′ frames and NR fields obtained by (arbitrarily) fixing the length of each $[vr]$ in $\underset{\sim}{y}$. $Channel_2$ is said to *satisfy* $\underset{\sim}{y}$ if, by replacing all S′ and I′ frames in $Channel_2$ by their NR fields, the resulting sequence equals an instance of $\underset{\sim}{y}$.

Given integers m and vr such that $m \ge 0$ and $0 \le vr < N$, the notation $[vr] .. [vr \ominus m]$ denotes the sequence $[vr], [vr \ominus 1], \ldots, [vr \ominus m]$ if $m > 0$, and the sequence $[vr]$ if $m = 0$. For convenience we assume that if $m > 0$, then $[vr \ominus m]$ is not empty.

Whenever the term [*Old_Ack_Sequence*] occurs in the assertions, it denotes any sequence (possibly empty) of entries of the form $[vr]$. We use [*Old_Ack_Sequence*] to refer to the $[vr]$ sequence obtained from the I′ and S′ frames in $Channel_2$ pertaining to a data connection that is in the process of being reset.

Lastly, (Final, NR) denotes an I′ or S′ frame whose F field equals 1 and whose receive sequence number equals NR.

Data Transfer Assertions.

The assertions A1–A6 listed below, in conjunction with the PF assertions, are inductively complete, and hence invariant (proof appears in [16, 18]. A1–A5 are concerned with conditions that hold during opening/closing of the data link; A6 is concerned with conditions of data transfer that hold when the data link is open.

A1. (a) $Mode_1$ = Open ⟹ $Mode_2$ = Open and no U′ frames in $Channel_1$
and no U′ frames in $Channel_2$ and *U_Response* = None
(b) $Mode_1$ = Closed ⟹ $Mode_2$ = Closed and $Channel_1$ is empty
and $Channel_2$ is empty and *U_Response* = None
(c) (I′, P, *Data*, NS) or (S′, P) in $Channel_1$ ⟹ $Mode_2$ = Open.

A2. *Poll_Timer* = Off ⟹ *U_Response* = None
and ($Mode_2$ = Open or $Mode_2$ = Closed).

A3. (U′, P, *Command*) in $Channel_1$
⟹ $Channel_1$ satisfies (U′, 1, *Command*), [*Old_Info_Sequence*]
and (*Command* = SARM ⟹ $Mode_1$ = Opening)
and (*Command* = DISC ⟹ $Mode_1$ = Closing)
and ($Mode_2$ = Open or $Mode_2$ = Closed) and *U_Response* = None
and $Channel_2$ satisfies [*Old_Ack_Sequence*].

A4. *U_Response* ≠ None ⟹ *Final_bit* = 1
and $Channel_1$ is empty and $Channel_2$ satisfies [*Old_Ack_Sequence*]

and ($U\_Response$ = UA $\Rightarrow$ ($Mode_1 = Mode_2$ = Opening
or $Mode_1 = Mode_2$ = Closing))
and ($U\_Response$ = DM $\Rightarrow$ ($Mode_1$ = Closing
and $Mode_2$ = Closed)).

A5. (U′, F, $Response$) in $Channel_2 \Rightarrow Channel_1$ empty
and(($Channel_2$ satisfies (U′, 1, DM), [$Old\_Ack\_Sequence$]
and $Mode_2$ = Closed and $Mode_1$ = Closing)
or ($Channel_2$ satisfies (U′, 1, UA), [$Old\_Ack\_Sequence$]
and $Mode_2$ = Closed and $Mode_1$ = Closing)
or ($Channel_2$ satisfies [0], (U′, 1, UA), [$Old\_Ack\_Sequence$]
and $Mode_2$ = Open, $VR = R = User\_out = 0$, $U\_Response$ = None
and $Mode_1$ = Opening)).

{A5 supplies the initial condition for the next assertion, A6, to hold when the data link is opened at P_1.}

A6. If $Mode_1$ = Open then ($Mode_2$ = Open and B1 and B2 and B3 and B4 and (B5 or B6)) holds, where B1-B6 are the assertions listed below.

B1. $Source[i] = Sink[i]$ for $0 \le i < User\_out \le R$.
{Data is transferred in sequence.}

B2. $A \le R \le S < A + N$.
{The number of outstanding blocks in P_1 is always less than N.}

B3. $A \bmod N = VA$, $S \bmod N = VS$, $R \bmod N = VR$.
{With B2 above, this asserts that the modulo N state variables point to the same data blocks in *Source* and *Sink*, as pointed to by A, S and R.}

B4. $Checkpoint\_Cycle$ = True $\Rightarrow VS \ominus VA > VCS \ominus VA$.
{The data blocks with sequence numbers VA, $VA \oplus 1, \ldots, VS \ominus 1$ are outstanding. When a checkpoint cycle is ongoing, then of these data blocks, the subset with sequence numbers VA, $VA \oplus 1, \ldots, VCS$ were outstanding when the Poll was sent.}

B5. (a) $Channel_1$ satisfies $\langle S - 1, VS \ominus 1 \rangle \, .. \, \langle S - n, VS \ominus n \rangle$ where $0 \le n \le S - R$.

{$Channel_1$ has a (possibly empty) sequence of I′ frames containing successive data blocks. If $n = S - R$ then $\langle S - n, VS \ominus n \rangle$ is the next data block expected by P_2. If $n < S - R$, then the data block next expected by P_2 has been lost, none of the I′ frames currently in $Channel_1$ will be accepted by P_2, and P_1 is not yet aware of the loss.}

(b) $Channel_2$ satisfies $[VR \, .. \, [VR \ominus m]$ where $0 \le m \le R - A$.

{$Channel_2$ contains a (possibly empty) sequence of successive receive sequence numbers. If $m > 0$ then $[VR \ominus m]$ consists of at least one I′ or S′ frame with its receive sequence number equal to $VR \ominus m$.}

(c) $Checkpoint\_Cycle$ and Poll in $Channel_1 \Rightarrow$ ($n = 0$ and $VCS = VS \ominus 1$)
or ($n > 0$ and Poll is with or to immediate left of $\langle S - i, VS \ominus i \rangle$
where $i = VS \ominus VCS$)

or ($n > 0$ and Poll is to right of $\langle S - n, VS \ominus n\rangle$
and $n = (VS \ominus VCS) - 1$).

{Relates position of a checkpoint Poll in $Channel_1$ to VCS.}

(d) $Checkpoint\_Cycle$ and $Final\_bit = 1$ and $(VR \ominus VA \le VCS \ominus VA) \Rightarrow S - n > R$.

{If checkpoint Poll has reached P_2 and all data up to the checkpoint has not been received at P_2, then data blocks are lost.}

(e) $Checkpoint\_Cycle$ and (Final, NR) in $Channel_2$
and $(NR \ominus NA \le VCS \ominus VA) \Rightarrow NR = VR$ and $S - n > R$.

{If checkpoint Final is in C_2 and its NR does not acknowledge all data blocks up to VCS, then data blocks are lost and the Final's NR equals VR.}

B6. {P_1 realized (some time ago) that P_2 was receiving out of sequence I′ frames, and has commenced (or is about to commence) retransmission of in sequence I′ frames.}

(a) $Channel_1$ satisfies $\langle S - 1, VS \ominus 1\rangle \,..\, \langle S - j, VS \ominus j\rangle$, $\langle s_0 - 1, vs_0 \ominus 1\rangle \,..\, \langle s_0 - k, vs_0 \ominus k\rangle$ where $j = S - A$, $A + N > s_0 > s_0 - k > R$, and $vs_0 = s_0 \bmod N$.

{The head of $Channel_1$ consists of one or more old out-of-sequence I′ frames $\langle s_0 - 1, vs_0 \ominus 1\rangle \,..\, \langle s_0 - k, vs_0 \ominus k\rangle$, followed by zero or more in-sequence I′ frames $\langle S - 1, VS \ominus 1\rangle \,..\, \langle S - j, VS \ominus j\rangle$.}

(b) $Channel_2$ satisfies $[VR]$.

{$Channel_2$ has zero or more receive sequence numbers, all equal to VR.}

(c) $R = A$.

{P_1 has determined the current value of R exactly. Hence $\langle S - j, VS \ominus j\rangle$, the first of the new I′ frames in $Channel_1$, is the one next expected by P_2.}

(d) (There is no Poll in $Channel_1$ with or to the right of $\langle s_0 - 1, vs_0 \ominus 1\rangle$) and $Final\_bit = 0$ and no Final in $Channel_2$.

{Since $\langle s_0 - 1, vs_0 \ominus 1\rangle$ was the last I′ frame sent before retransmission was initiated through checkpointing, the Poll is either not outstanding or has been sent after $\langle s_0 - 1, vs_0 \ominus 1\rangle$ was sent.}

(e) $Checkpoint\_Cycle$ and Poll in $Channel_1 \Rightarrow$ Poll is with or to the left of $\langle S - i, VS \ominus i\rangle$ where $i = VS \ominus VCS$.

{Relates position of a checkpoint Poll in $Channel_1$ to VCS.}

3.4 Image Protocol for P_2 to P_1 Data Transfer

We now consider the function of one-way data transfer from P_2 to P_1. Two desirable properties of the HDLC protocol concerning this function may be stated as follows:

If $Mode_2 = Mode_1 =$ Open then

1. $Sink_1[i] = Source_2[i]$ for $0 \le i < User\_out_1$
2. $0 \le A_2 \le S_2 < A_2 + N$

The first property states that data is transferred in sequence; the second that the maximum number of outstanding data blocks (hence the minimum storage requirement) at P_2 is $N - 1$. These properties can be stated using only the variables *Mode, Sink,* and *User_out* at P_1, and *Mode, Source, S* and *A* at P_2. As in the case of the P_1 to P_2 data transfer, we can construct an image protocol using the variables *Poll_Timer, $Poll_Timer, Poll_Retry_Count, VR, R,* and *Local_RStatus* at P_1, and *Final_bit, $Response_Time, U_Response, User_in, VS, VA, VCS, Checkpoint_Cycle* and *Remote_RStatus* at P_2. (This image protocol is not well-formed.)

The images of message types sent by P_1 can be defined as follows. The image of message type (U, *P, Command*) is (U′, *P, Command*). The image of (I, *P, Data, NS, NR*) is defined as (I′, *P, NR*), where I′ is a new message type corresponding to an I frame in the HDLC protocol. The image of (S, *P, RStatus, NR*) is (S′, *P, RStatus NR*).

Next we consider the images of the HDLC message types sent by P_2. The image of (U, *F, Response*) is (U′, *F, Response*). The image of (I, *F, Data, NS, NR*) is (I′, *F, Data, NS*), where I′ is a new message type corresponding to an I frame of the HDLC protocol. The image of (S′, *F, RStatus, NR*) is (S′, *F*), where S′ is a new message type corresponding to an S frame of the HDLC protocol.

Thus, in the image protocol, (U′, *P, Command*), (I′, *P, NR*) and (S′, *F, RStatus, NR*) are the message types sent by P_1, and (U′, *F, Response*), (I′, *F, Data, NS*), and (S′, *F*) are the message types sent by P_2.

The events of P_1 and P_2 are shown in Tables X and XI, respectively. The channel and time events of this image protocol are as in Tables IV and V. The initial state of this image protocol is the same as that of the HDLC protocol.

3.4.1. *Safety Properties.* For this image protocol, we have verified the desirable safety properties of the projected function. The following assertions, in conjunction with the PF assertions, are inductively complete and hence invariant. The notation used in these assertions is similar to the notation used in the assertions for the image protocol of P_1 and P_2 data transfer (with the role of P_1 and P_2, $Channel_1$ and $Channel_2$, and Poll and Final interchanged).

The assertions C1–C6 listed below hold at all times. C1–C5 are concerned with conditions that hold during opening/closing of the data link; C6 is concerned with conditions of data transfer that hold when the data link is open.

C1. (a) $Mode_1$ = Open $\Rightarrow$ $Mode_2$ = Open and no U′ frames in $Channel_1$
    and no U′ frames in $Channel_2$ and *U_Response* = None
  (b) $Mode_1$ = Closed $\Rightarrow$ $Mode_2$ = Closed and $Channel_1$ is empty
    and $Channel_2$ is empty and *U_Response* = None
  (c) (I′, *P, NR*) or (S′, *P, RStatus, NR*) in $Channel_1$ $\Rightarrow$ $Mode_2$ = Open.

C2. *Poll_Timer* = Off $\Rightarrow$ *U_Response* = None
    and ($Mode_2$ = Open or $Mode_2$ = Closed).

C3. (U′, *P, Command*) in $Channel_1$
  $\Rightarrow$ $Channel_1$ satisfies (U′, 1, *Command*), [*Old_Ack_Sequence*]
    and (*Command* = SARM $\Rightarrow$ $Mode_1$ = Opening)
    and (*Command* = DISC $\Rightarrow$ $Mode_1$ = Closing)

and ($Mode_2$ = Open or $Mode_2$ = Closed) and $U\_Response$ = None
and $Channel_2$ satisfies [$Old\_Info\_Sequence$].

C4. $U\_Response \neq$ None $\Rightarrow$ $Final\_bit = 1$
and $Channel_1$ is empty and $Channel_2$ satisfies [$Old\_Info\_Sequence$]
and ($U\_Response$ = UA $\Rightarrow$ ($Mode_1 = Mode_2$ = Opening
or $Mode_1 = Mode_2$ = Closing))
and ($U\_Response$ = DM $\Rightarrow$ ($Mode_1$ = Closing
and $Mode_2$ = Closed)).

C5. (U′, F, $Response$) in $Channel_2 \Rightarrow Channel_1$ empty
and (($Channel_2$ satisfies (U′, 1, DM), [$Old\_Info\_Sequence$]
and $Mode_2$ = Closed and $Mode_1$ = Closing)
or ($Channel_2$ satisfies (U′, 1, UA), [$Old\_Info\_Sequence$]
and $Mode_2$ = Closed and $Mode_1$ = Closing)
or ($Channel_2$ satisfies $\langle S-1, VS \ominus 1\rangle \ldots \langle 0, 0\rangle$, (U′, 1, UA),
[$Old\_Info\_Sequence$]
and $Mode_2$ = Open, $VA = A = 0$, $0 \leq VS = S < N$,
$Checkpoint\_Cycle$ = False, $U\_Response$ = None
and $Mode_1$ = Opening)).

{C5 supplies the initial condition for the next assertion, C6, to hold when the data link is opened at P_1.}

C6. If $Mode_1$ = Open then ($Mode_2$ = Open and D1 and D2 and D3 and D4 and (D5 or D6)) holds, where D1–D6 are the assertions listed below.

D1. $Source[i] = Sink[i]$ for $0 \leq i < User\_out \leq R$.

D2. $A \leq R \leq S < A + N$.

D3. $A \bmod N = VA$, $S \bmod N = VS$, $R \bmod N = VR$.

D4. $Checkpoint\_Cycle$ = True $\Rightarrow VS \ominus VA > VCS \ominus VA$.

D5. (a) $Channel_2$ satisfies $\langle S-1, VS \ominus 1\rangle \ldots \langle S-n, VS \ominus n\rangle$
where $0 \leq n \leq S - R$.

(b) $Channel_1$ satisfies $[VR] \ldots [VR \ominus m]$
where $0 \leq m \leq R - A$.

(c) $Checkpoint\_Cycle$ and Final in $Channel_2 \Rightarrow$ ($n = 0$ and $VCS = VS \ominus 1$)
or ($n > 0$ and Final is with or to immediate left of $\langle S-i, VS \ominus i\rangle$
where $i = VS \ominus VCS$)
or ($n > 0$ and Final is to right of $\langle S-n, VS \ominus n\rangle$
and $n = (VS \ominus VCS) - 1$).

(d) $Checkpoint\_Cycle$ and $Poll\_Timer$ = Off
and $(VR \ominus VA \leq VCS \ominus VA) \Rightarrow S - n > R$.

(e) $Checkpoint\_Cycle$ and (Poll, NR) in $Channel_1$
and $(NR \ominus VA \leq VCS \ominus VA) \Rightarrow NR = VR$ and $S - n > R$.

D6. (a) $Channel_2$ satisfies $\langle S-1, VS \ominus 1\rangle \ldots \langle S-j, VS \ominus j\rangle$,
$\langle s_0 - 1, vs_0 \ominus 1\rangle \ldots \langle s_0 - k, vs_0 \ominus k\rangle$

where $j = S - A$, $A + N > s_0 > s_0 - k > R$, and $vs_0 = s_0 \bmod N$.

(b) *Channel*$_1$ satisfies $[VR]$.

(c) $R = A$.

(d) (There is no Final in *Channel*$_2$ with or to right of $\langle s_0 - 1, vs_0 \ominus 1\rangle$) and *Poll_Timer* $\neq$ Off and no Poll in *Channel*$_1$.

(e) *Checkpoint_Cycle* and Final in *Channel*$_2$
$\Rightarrow$ Final is with $\langle S - j, VS \ominus j\rangle$.

3.5 A Proposed Modification to HDLC

We observed above that HDLC protocol cannot be considered to be well-structured, since it does not have well-formed image protocols that are significantly smaller than the HDLC protocol itself for all of its functions. We will now introduce a minor modification to the I message in the HDLC protocol. This modification allows us to obtain small well-formed image protocols for each of the one-way data transfer functions.

Our modification consists of adding an *RStatus* field to the HDLC I message type. In place of the message type (I, *P*, *Data*, *NS*, *NR*) sent by P$_1$, we will use the message type (IM, *P*, *Data*, *NS*, *NR*, *RStatus*), where IM (standing for I modified) is the name of the message type. In place of the message type (I, *F*, *Data*, *NS*, *NR*) sent by P$_2$, we will use the message type (IM, *F*, *Data*, *NS*, *NR*, *RStatus*). Note that the *RStatus* field can be implemented using a field of one bit.

The usage of this IM message type is similar to the usage of the HDLC I message type, except for the following difference. The *P* field being set to 1 does not indicate any flow control information; instead, the *RStatus* field is used to convey flow control information exactly as in an S frame. The events of this modified HDLC protocol system are shown in Tables XII and XIII. Note that the only difference between this modified HDLC protocol and the original HDLC protocol (Tables I and II) is that Send_I and Rec_I have been replaced by Send_IM and Rec_IM, respectively. This modified HDLC protocol possesses small well-formed image protocols for each of its functions [16] and can be considered to be well-structured.

In particular, the image protocol obtained for connection management in Section 3.2 (Tables VI and VII) is still valid and well-formed (where I′ is now the image of IM and S).

In the image protocol for one-way data transfer from P$_1$ to P$_2$, we have the same entity variables as before (Section 3.3). The message type U is not changed in the image protocol (as before). The image of message types IM and S sent by P$_1$ are I′ and S′ respectively, as already defined. The message types IM and S sent by P$_2$ are both projected onto the same image message type S (i.e., the *Data* and *NS* fields in IM are deleted). The events of P$_1$ (P$_2$) in the image protocol are exactly as shown in Table VIII (Table IX), except that Rec_I′(Send_I′) is missing. This image protocol can easily be shown to be well-formed [16].

The image protocol for one-way data transfer from P$_2$ to P$_1$ can be similarly constructed. The entity variables of P$_1$ and P$_2$ in the image protocol are as defined in Section 3.4. The message types IM and S sent by P_1 are both projected onto the same message type S. The message types IM and S sent by P_2 have the images I′ and S′ respectively as already defined. The events of P_1 (P_2) in the image protocol are exactly as shown in Table X (Table XI), except that Send_I′ (Rec_I′) is missing. This image protocol can easily be shown to be well-formed.

The connection management assertions obtained in Section 3.2.1, and the data transfer assertions obtained in Sections 3.3.1 and 3.4.1 continue to be invariant for the image protocols of this modified HDLC protocol. In addition, because these image protocols are well-formed, they are faithful to the modified HDLC protocol.

4. CONCLUSION

We have specified a version of the HDLC protocol using an event-driven process model, and verified it using the method of projections. The verification serves as a rigorous exercise in demonstrating the applicability of our method to the analysis of real-life protocols.

The HDLC protocol specified is based upon the Asynchronous Response Mode (ARM) of operation between two protocol entities, 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, cyclic sequence numbers, sliding windows of size N for data transfers, and ready/not ready messages for flow control.

The HDLC protocol has two characteristics found in most real-life communication protocols. First, the HDLC protocol is a time-dependent system, that is, HDLC operates under real-time constraints that are important not only for the protocol's performance efficiency but also for its correct logical behavior. Such time-dependent behavior cannot be handled by liveness assertions of the temporal logic variety. By including time variables and time events in our protocol model, we specify the HDLC time-dependent behavior in terms of safety assertions.

Second, HDLC is a multifunction protocol. It implements three distinguishable functions: connection management, and one-way data transfers between two protocol entities. Using the method of projections, we have constructed for each function an image protocol containing only those portions of the HDLC protocol that are needed to verify the desired correctness properties of that function. In each case, an inductively complete assertion implying the desired behavior was obtained.

Of the three image protocols obtained, only the connection management image protocol is well-formed. In order to construct a well-formed image protocol for one-way data transfer, almost the entire HDLC protocol has to be included in the image. This is due to dependencies in the two data transfer functions of HDLC. Thus, we say that the HDLC protocol as currently specified is not well-structured. We then introduced a minor modification to the HDLC protocol, in the form of an additional one-bit flow control field in the HDLC I frame format. With this modification, small well-formed image protocols can be constructed for each of the HDLC functions of interest. Our modified version of HDLC can be considered as as well-structured protocol.

APPENDIX A

PROOF OF PF ASSERTIONS.

Initially, *Poll_bit* = 0, *Poll_Timer* = Off, *Final_bit* = 0, *$Response_Time* = Off, and the channels are empty. Hence, PF2 holds nonvacuously, while the rest of the PF assertions hold vacuously. We will next show that the PF assertions are true after the occurrence of any event, provided that they are true before the event occurrence. This will then establish the PF assertions as a system invariant.

We first introduce some notation that is used in the proof. The name of a variable will be used to denote its value before the occurrence of an event. We will now consider each of the events of the image protocol system and show that the PF assertions are not violated by any of them.

Entity events of P_1. User_req_conn and User_req_disc do not affect any of the PF assertions.

In order to send a Poll (using either Send_U′ or Send_I′), the condition *Poll_bit* = 1 must hold. From PF1 and PF2, this means that there is no Poll in $Channel_1$, no Final in $Channel_2$, and *Final_bit* = 0. After the event, *Poll_Timer* = *$Poll_Timer* = 0, and there is a single (Poll, *$Age*) in $Channel_1$ with *$Age* = 0 (Poll is either a U′ or I′ frame). This establishes PF3 after the event. The other PF assertions are vacuously true.

Sending a non-Poll message does not affect the PF assertions.

When a Final message (either a U′ or an I′ frame) is received, from PF5 we have the following: there is no Poll in $Channel_1$, *Final_bit* = 0 and $Channel_2$ has exactly one Final. Hence after the event occurrence, PF2 holds nontrivially, while the other PF assertions hold vacuously.

Reception of a non-Final message does not affect the PF assertions.

Request_Poll event makes PF1 hold nontrivially. The other PF assertions are not affected and continue to hold.

Poll_Timeout event can occur only when *Poll_Timer* ≥ *PollTimeoutValue.* Since *PollTimeoutValue* > 1 + (1 + *a*)($MaxDelay_1$ + *MaxResponseTime* + $MaxDelay_2$), and (from the accuracy axiom) |*Poll_Timer* - *$Poll_Timer*| ≤ 1 + *a* *$Poll_Timer*, we have *$Poll_Timer* > $MaxDelay_1$ + *MaxResponseTime* + $MaxDelay_2$. If any one of PF3, PF4 or PF5 held nonvacuously before the Poll_Timeout event occurrence, that would imply either that channel C_1 contains an over-age message (*$Age* > $MaxDelay_1$ in PF3), or that entity P_2 waits too long to send a Final (*$Response_Time* > *MaxResponseTime* in PF4), or that channel C2 contains an over-age message (*$Age* > $MaxDelay_2$ in PF5). Because of the local time axioms of these components, none of this can occur. Hence, PF3, PF4 and PF5 will hold vacuously before the Poll_Timeout event. Hence PF2 holds nonvacuously after the Poll_Timeout event.

Entity events of P_2. The reception of Poll (either a U′ or an I′ frame) means that PF3 held nonvacuously. Also, from the local time axiom for channel C_1 we have that *$Poll_Timer* = *$Age* ≤ $MaxDelay_1$. After the reception of the Final, *Final_bit* = 1 and *$Response_Time* = 0. Hence PF4 holds nonvacuously after the event. The other PF assertions hold vacuously.

In order to send a Final message, *Final_bit* must equal 1. Hence PF4 holds nonvacuously. Also, from the local time axiom of entity P_2 (*$Response_Time* ≤

MaxResponseTime), we have that $\$Poll\_Timer \leq MaxDelay_1 + MaxResponseTime$. After the event occurrence, $Final\_bit = 0$, $\$Response\_Time$ = Off, and a (Final, $\$Age$) message with $\$Age = 0$ has been placed in the $Channel_2$. Hence, PF5 holds nonvacuously after the event occurrence, while the other PF assertions hold vacuously.

Reception of a non-Poll message or sending of a non-Final message does not affect the PF assertions.

Channel Events. Recall that the only channel event we have is a loss of the first message in any channel. The only effect of this message loss on the PF assertions is to transform PF3 (or PF4) from nonvacuously true to vacuously true. Hence the channel events do not violate the PF assertions.

Time Events. The occurrence of Poll_Timer_Tick does not affect the PF assertions since *Poll_Timer* is never set to Off by this event. The occurrence of the global time event increments by 1 tick both the left and right hand sides of the expressions involving *$Poll_Timer* in PF3, PF4, and PF5. Hence, none of the PF assertions are violated by the time events. □

APPENDIX B

Table I. Events of Primary HDLC Entity P_1

Event Name	Enabling Condition	Action
1. User_req_conn	Mode ≠ Opening **and** Mode ≠ Closing	Mode := Opening
2. User_req_disc	Mode = Open	Mode := Closing

3. User_puts_data	Mode = Open
	and (User_in-A<Sbufsize)
		{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,R,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	first(Channel$_2$) = I	get(Channel$_2$, (I,F,Data,NS,NR));
		if Mode = Open **then**
		   begin
		      DATA_NS_RECEIVED;
		      NR_RECEIVED;
		      if F = 1 **then**
		         begin
		            CHECKPOINT_RECEIVED;
		            FINAL_RECEIVED;
		            Remote_RStatus := RR
		         end
		   end

12. Rec_S	first(Channel$_2$) = S	get(Channel$_2$,(S,F,RStatus,NR));
		if Mode = Open **then**
		   begin
		      Remote_RStatus := RStatus;
		      NR_RECEIVED;
		      if F = 1 **then**
		         begin
		            CHECKPOINT_RECEIVED;
		            FINAL_RECEIVED
		         end
		   end

Table II. Events of Secondary HDLC Entity P_2

Event Name	Enabling Condition	Action
1. User_puts_data	Mode = Open and (User_in-A<SbufSize)	{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	get(Channel$_1$, (U,P,Command)); **if** U_Response ≠ UA **then** **begin** **if** Command = SARM **then** **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 **end**
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	get(Channel$_1$, (I,P,Data,NS,NR)); **if** U_Response ≠ UA **then** **begin** **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; Remote_RStatus := RR **end** **end** **end**
8. Rec_S	first(Channel$_1$) = S	get(Channel$_1$, (S,P,RStatus, NR)); **if** U_Response ≠ UA **then** **begin** **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** **end**

Table III. Details of Code Segments Used in Tables

```
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 = NS 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 ≠ VA 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 IV. Events of Channel C_i for the Protocol System

Event Name	Enabling Condition	Action
Message_Loss	Channel_i is not empty	Delete the first message in Channel_i

Table V. 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)

Table VI. Events of P_1 in the Image Protocol for Connection Management

Event Name	Enabling Condition	Action
1. User_req_conn	(same as in Table 1)	
2. User_req_disc	(same as in Table 1)	
3. Send_U'	(same as in Table 1)	
4. 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 Mode := Open;
		if F = 1 **then** FINAL_RECEIVED
5. Poll_Timeout	(same as in Table 1)	
6. Request_Poll	(same as in Table 1)	
7. Send_I'	Mode = Open	put(Channel$_1$, (I',Poll_bit));
		if Poll_bit = 1 **then** POLL_SENT
8. Rec_I'	first(Channel$_2$) = I'	get(Channel$_2$, (I',F));
		if Mode = Open **then**
		if F = 1 **then** FINAL_RECEIVED

Table VII. Events of P_2 in the Image Protocol for Connection Management

Event Name	Enabling Condition	Action
1. Rec_U'	(same as in Table 2)	
2. 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 Mode := Open;
		if Final_bit = 1
		then FINAL_SENT
3. Send_I'	Mode = Open	put(Channel$_2$, (I',Final_bit));
		if Final_bit = 1 **then** FINAL_SENT
4. Rec_I'	first(Channel$_1$) = I'	get(Channel$_1$, (I',P));
		if U_Response ≠ UA **then**
		begin
		if Mode = Closed
		then U_Response := DM;
		if (Mode = Open **and** P = 1)
		then POLL_RECEIVED
		end

Table VIII. Events of P_1 in the Image Protocol for the HDLC P_1 to P_2 Data Transfer

Event Name	Enabling Condition	Action
1. User_req_conn	(same as in Table 1)	
2. User_req_disc	(same as in Table 1)	
3. User_puts_data	(same as in Table 1)	
4. Send_U'	(same as in Table 1)	
5. 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 **end**; **if** F = 1 **then** FINAL_RECEIVED
6. Poll_Timeout	(same as in Table 1)	
7. Request_Poll	(same as in Table 1)	
8. Send_I'	Mode = Open **and** VS$\ominus$VA < N-1 **and** S < User_in **and** Remote_RStatus = RR	put($Channel_1$, (I',Poll_bit,Source[S],VS)); VS := VS $\oplus$ 1; S := S + 1; **if** Poll_bit = 1 **then** **begin** CHECKPOINT_SENT; POLL_SENT **end**
9. Send_S'	Mode = Open	put($Channel_1$, (S',Poll_bit)); **if** Poll_bit = 1 **then** **begin** CHECKPOINT_SENT; POLL_SENT **end**
10. Rec_I'	first($Channel_2$) = I'	get($Channel_2$, (I',F,NR)); **if** Mode = Open **then** **begin** NR_RECEIVED; **if** F = 1 **then** **begin** CHECKPOINT_RECEIVED; FINAL_RECEIVED; Remote_RStatus := RR **end** **end**
11. Rec_S'	(same as in Table 1)	

Table IX. Events of P_2 in the Image Protocol for the HDLC P_1 to P_2 Data Transfer

```
Event Name          Enabling Condition            Action

1. User_gets_data                (same as in Table 2)

2. Rec_U'                        (same as in Table 2)

3. 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
                                                     end;
                                                  if Final_bit = 1 then FINAL_SENT

4. Send_I'          Mode = Open                   put(Channel$_2$, (I',Final_bit,VR));
                    and not(Final_bit = 1         if Final_bit = 1 then FINAL_SENT
                        and Local_RStatus=RNR)

5. Send_S'          Mode = Open                   put(Channel$_2$, (S',Final_bit,Local_RStatus,VR));
                                                  if Final_bit = 1 then FINAL_SENT

6. Rec_I'           first(Channel$_1$) = I'       get(Channel$_1$, (I',P,Data,NS));
                                                  if U_Response ≠ UA then
                                                     begin
                                                        if Mode = Closed
                                                           then U_Response := DM;
                                                        if Mode = Open then
                                                           begin
                                                              DATA_NS_RECEIVED;
                                                              if P = 1 then POLL_RECEIVED
                                                           end
                                                     end

7. Rec_S'           first(Channel$_1$) = S'       get(Channel$_1$, (S',P));
                                                  if U_Response ≠ UA then
                                                     begin
                                                        if Mode = Closed
                                                           then U_Response := DM;
                                                        if Mode = Open and P = 1
                                                           then POLL_RECEIVED
                                                     end
```

Table X. Events of P_1 in the Image Protocol for the HDLC P_2 to P_1 Data Transfer

Event Name	Enabling Condition	Action
1. User_req_conn	(same as in Table 1)	
2. User_req_disc	(same as in Table 1)	
3. User_gets_data	(same as in Table 1)	
4. Send_U'	(same as in Table 1)	

5. 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_REC_VARIABLES
		   end;
		if F = 1 **then** FINAL_RECEIVED

6. Poll_Timeout	(same as in Table 1)	
7. Request_Poll	(same as in Table 1)	

8. Send_I'	Mode = Open
	and not(Poll_bit = 1
	   and Local_RStatus=RNR)
		put(Channel$_1$, (I',Poll_bit,VR));
		if Poll_bit = 1 **then** POLL_SENT

9. Send_S'	Mode = Open	put(Channel$_1$, (S',Poll_bit,Local_RStatus,VR));
		if Poll_bit = 1 **then** POLL_SENT

10. Rec_I'	first(Channel$_2$) = I'	get(Channel$_2$, (I',F,Data,NS));
		if Mode = Open **then**
		   begin
		      DATA_NS_RECEIVED;
		      if F = 1 **then** FINAL_RECEIVED
		   end

11. Rec_S'	first(Channel$_2$) = S'	get(Channel$_2$, (S',F));
		if Mode = Open **then**
		   if F = 1 **then** FINAL_RECEIVED

Table XI. Events of P_2 in the Image Protocol for the HDLC P_2 to P_1 Data Transfer

Event Name	Enabling Condition	Action
1. User_puts_data	(same as in Table 2)	
2. Rec_U'	(same as in Table 2)	
3. 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 **end**; **if** Final_bit = 1 **then** FINAL_SENT
4. Send_I'	Mode = Open **and** VS ⊖ VA < N-1 **and** S < User_in **and** Remote_RStatus = RR	put(Channel$_2$, (I',Final_bit,Source[S],VS)); VS := VS ⊕ 1; S := S + 1; **if** Final_bit = 1 **then** **begin** CHECKPOINT_SENT; FINAL_SENT **end**
5. Send_S'	Mode = Open	put(Channel$_2$, (S',Final_bit)); **if** Final_bit = 1 **then** **begin** CHECKPOINT_SENT; FINAL_SENT **end**
6. Rec_I'	first(Channel$_1$) = I'	get(Channel$_1$, (I',P,NR)); **if** U_Response ≠ UA **then** **begin** **if** Mode = Closed **then** U_Response := DM; **if** Mode = Open **then** **begin** NR_RECEIVED; **if** P = 1 **then** **begin** CHECKPOINT_RECEIVED; FINAL_RECEIVED; Remote_RStatus := RR **end** **end** **end**
7. Rec_S'	(same as in Table 2)	

Table XII. Events of P_1 in the Modified HDLC Protocol

Event Name	Enabling Condition	Action
1. Send_IM	Mode = Open	put(Channel$_1$,(IM,Poll_bit,Source[S],VS,VR,Local_RStatus));
	and VS$\ominus$VA < N-1	VS := VS $\oplus$ 1; S := S + 1;
	and S < User_in	if Poll_bit = 1 then
	and Remote_RStatus = RR	begin
		CHECKPOINT_SENT;
		POLL_SENT
		end
2. Rec_IM	first(Channel$_2$) = IM	get(Channel$_2$, (IM,F,Data,NS,NR,RStatus));
		if Mode = Open then
		begin
		DATA_NS_RECEIVED;
		NR_RECEIVED;
		Remote_RStatus := RStatus;
		if F = 1 then
		begin
		CHECKPOINT_RECEIVED;
		FINAL_RECEIVED
		end
		end

[User_req_conn, User_req_disc, User_puts_data, User_gets_data, Send_U, Rec_U, Poll_Timeout, Request_Poll, Send_S, Rec_S, are the same as in Table 1]

Table XIII. Events of P_2 in the Modified HDLC Protocol

Event Name	Enabling Condition	Action
1. Send_IM	Mode = Open	put(Channel$_2$,(IM,Final_bit,Source[S],VS,VR,Local_RStatus));
	and VS$\ominus$VA < N-1	VS := VS $\oplus$ 1; S := S + 1;
	and S < User_in	if Final_bit = 1 then
	and Remote_RStatus = RR	begin
		CHECKPOINT_SENT;
		FINAL_SENT
		end
2. Rec_IM	first(Channel$_1$) = IM	get(Channel$_1$, (IM,P,Data,NS,NR,RStatus));
		if U_Response $\neq$ UA then
		begin
		if Mode = Closed
		then U_Response := DM;
		if Mode = Open then
		begin
		DATA_NS_RECEIVED;
		NR_RECEIVED;
		Remote_RStatus := RStatus;
		if P = 1 then
		begin
		CHECKPOINT_RECEIVED;
		POLL_RECEIVED
		end
		end
		end

[User_puts_data, User_gets_data, Send_U, Rec_U, Send_S, Rec_S, are the same as in Table 2]

REFERENCES

1. BOCHMANN, G.V. Finite state description of communication protocols. *Comput. Networks 2*, (Oct. 1978), 361–372.
2. BOCHMANN, G.V., AND CHUNG, R.J. A formalized specification of HDLC classes of procedures. In *Conf. Rec. National Telecommunication Conference*, (Los Angles, Dec. 1977), IEEE, New York, pp. 519–530.

3. BOCHMANN, G.V., AND C.A. SUNSHINE, Formal methods in communication protocol design. *IEEE Trans. Commun. COM 28*, 4 (April 1980), 624-631.
4. BRAND, D. AND JOYNER, W.H. Verification of HDLC. *IEEE Trans. Commun. COM 30*, 5 (May 1982), 1136-1142.
5. DIJKSTRA, E.W. *A Discipline of Programming*. Prentice-Hall, Englewood, Cliffs N.J., 1976.
6. HAILPERN, B.T. AND OWICKI, S.S. Verifying network protocols using temporal logic. Tech. Rep. 192, Computer Systems Laboratories, Stanford University, Stanford, Calif., June, 1980.
7. INTERNATIONAL STANDARDS ORGANIZATION. *Data Communincations—HDLC Procedures—Frame Structure*. Ref. No. ISO 3309, International Standards Organization, Geneva, Switzerland, 1979.
8. INTERNATIONAL STANDARDS ORGANIZATION. *Data Communications—HDLC Procedures—Elements of Procedures*. Ref. No. ISO 4335, International Standards Organization, Geneva, Switzerland, 1979.
9. INTERNATIONAL STANDARDS ORGANIZATION. *Data Communications—HDLC Unbalanced Classes of Procedures*. Ref. No. ISO 6159, International Standards Organization, Geneva, Switzerland, 1980.
10. KUROSE, J. The specification and verification of a connection establishment protocol using temporal logic. In *Proc. 2nd Int. Workshop on Protocol Specification, Testing and Verification*, (Idyllwild, Ca., May 17-20, 1982), IFIP, New York, pp. 43-62.
11. LAM, S.S. AND SHANKAR, A.U. Protocol projections: A method for analyzing communication protocols. In *Conf. Rec. National Telecommunications Conference*, (New Orleans, Nov. 1981), IEEE, New York.
12. LAM, S.S. AND SHANKAR, A.U. Verification of communication protocols via protocol projections. In *Proc INFOCOM '82*, (Las Vegas, March 30-April 1, 1982), IEEE, New York, pp. 229-237.
13. LAM, S.S., AND SHANKAR, A.U. *Protocol Verification via Projections*. Tech. Rep. 207, Dept. of Comput. Sci., Univ. of Texas at Austin, Aug., 1982. To appear in *IEEE Trans. Softw. Eng.*
14. RAZOUK, R. Modeling X.25 using the graph model of behavior. In *Proc. 2nd Int. Workshop on Protocol Specification, Testing and Verification*, (Idyllwild, Ca., May 17-20, 1982). IFIP, New York, pp. 197-214.
15. SCHWARTZ, R.L. AND MELLIAR-SMITH, P.M. From state machines to temporal logic: Specification methods for protocol standards. *IEEE Trans. Commun. COM-30*, 12 (Dec. 1982), 2486-2496.
16. SHANKAR, A.U. Analysis of communication protocols via protocol projections. Ph.D. thesis, Dept. of Electrical Engineering, University of Texas at Austin, Austin, Tx., December, 1982.
17. SHANKAR, A.U. AND LAM, S.S. On time-dependent communication protocols and their projections. In *Proc. 2nd Int. Workshop on Protocol Specification, Testing and Verification*, (Idyllwild, Ca., May 17-20, 1982), IFIP, New York, pp. 215-235.
18. SHANKAR, A.U. AND LAM, S.S. *An HDLC Protocol Specification and its Verification using Image Protocols*. Tech. Rep. 212, Dept. of Computer Sciences, University of Texas at Austin, September, 1982 (first version).
19. SHANKAR, A.U. AND LAM, S.S. Specification and verification of an HDLC protocol with ARM connection management and full-duplex data transfer. In *Proc. ACM SIGCOMM '83 Symposium*, (Austin, Tx., March 8-9, 1983), ACM, New York, pp. 38-48.
20. SHANKAR, A.U. AND LAM, S.S. *Application of Projections to a Structured Model of Communication Protocols*. Tech. Rep. 214, Dept. of Computer Sciences, University of Texas at Austin, 1983 (in preparation).
21. SLOAN, L. Mechanisms that enforce bounds on packet lifetimes. Presented at *ACM SIGCOMM '83 Symposium*, University of Texas at Austin, March 1983. *ACM Trans. Comput. Syst.* 1, 4 (Nov. 1983), 311-330 (this issue).
22. STENNING, N.V. A data transfer protocol. *Comput. Networks 1* (Sept. 1976), 99-110.
23. ZIMMERMANN, H., OSI Reference Model— The ISO model of architecture for open systems interconnection. *IEEE Trans. Commun. COM-28*, 4 (April 1980), 425-432.

Received September 1982; revised May 1983, accepted July 1983