I. INTRODUCTION

Programs that implement computer communication protocols can exhibit extremely complicated behavior, because they must cope with asynchronous computing agents and the possibility of failures in the agents and in the communication medium. A survey of the literature in the area of protocol verification can be found in Bochmann and Sunshine [20]. Most previous approaches to verifying network protocols have been based upon reachability arguments for finite-state models of the protocols. This technique has the advantage of being easily automated. It encounters difficulties, however, as the state space of the protocol becomes large. For example, finite-state models present difficulties in dealing with properties related to correct data transfer, because representing each value to be transmitted can make the state space extremely large, and possibly infinite. Bochmann and Sunshine [20] present some techniques for reducing the state space: partial verification (not proving all aspects correct), combining or ignoring certain states, using assertions to classify states, and focusing search (not checking all paths). All these techniques involve ignoring some states or using some non-finite-state tool (which cannot, in general, be automated). In contrast, the approach described here models a protocol as a parallel program, and correctness proofs follow the Floyd-Hoare style of program verification. Logical assertions attached to the program abstract information from the representation of the state and allow reasoning about classes of states. This avoids the combinatorial explosion, and the length of the proof need not grow unmanageably as the protocol size increases. (Our technique is not automated, but many automatic Floyd-Hoare verifiers are being developed by others.)

In this approach, the network/protocol system is modeled by a set of interacting modules that represent logical units of the system, such as the communication medium, transmitter, and receiver. There are two kinds of modules to be considered: processes and monitors. A process is an active program component, and a monitor is a data abstraction with synchronization [3], [8]. We exploit the modularity of the system model in the construction of proofs. At the lowest level, the properties of processes and monitors are verified by examination of their code. In constructing the system proof we use these verified properties and can ignore the internal structure of the module implementations. For example, buffers are an abstract data type that can be implemented in many ways. Any implementation meeting the requirements of the data type may be used in the protocol without affecting the correctness proof of the rest of the system.

Two kinds of properties, safety and liveness, are important for parallel systems. Safety properties have the form “bad things will not happen.” They are analogous to partial correctness and are expressed by invariant assertions which must be satisfied by the system state at all times. Safety properties are often expressed in terms of auxiliary variables that record the history of the interactions of the modules. Because auxiliary variables are not implemented, they can record histories of unbounded length and are an important element in our proofs. Safety proofs are constructed as follows. One first verifies the invariants of the lowest level modules directly from the code. One then shows that in conjunction these invariants imply the invariants of larger components, ultimately arriving at a proof of the invariant of the whole system.

Liveness properties have the form “good things will happen.” They include termination requirements in sequential programs and recurrent properties in nonterminating programs like operating systems. Until recently, there has been little work on the verification of liveness properties, other than sequential termination. Because liveness refers to the future occurrence of a desired state, conventional logical formulas, which only refer to a single state, are inadequate for expressing and reasoning about liveness. To deal with liven-
ness, we use the notation of temporal logic [16], which provides operators for making assertions about future program states. Temporal formulas expressing liveness properties are called commitments. Commitments are verified in the same modular style as invariants: one first verifies the commitments of the lowest level modules directly from the code, and then shows that, in conjunction, they imply the commitments of the higher level modules.

The rest of this paper is organized as follows. In Section II we briefly review some of the tools we will use: temporal logic, history variables, and modular specifications. In Sections III and IV we discuss the verification of two protocols, Stening's protocol [19] and the alternating bit protocol [11]. Our emphasis in these sections is on the specifications of each module, and their composition to imply the system specifications. We will not perform the lowest level verification of invariants and commitments from module code, although we will indicate how it could be performed. That phase of the verification is carried out by Hailpern [7].

II. VERIFICATION TOOLS

A. Temporal Logic

Temporal logic provides operators for reasoning about the past and the future, although we will only need the operators for the future. In the context of program verification, the “future” is a program computation, that is, a sequence of states that could arise during program execution. Informally, the first state in a computation represents the present, and subsequent states represent the future. Computations are not restricted to starting at the beginning of the program, so a “future” state in one computation may be the “present” state in another.

The version of temporal logic we will use was developed by Pnueli [5], [16]-[18], and is further described by Lamport [10]. A discussion of its use in program verification can be found in Owicki and Lamport [15]. Its two basic operators are $\Box$ (henceforth) and $\Diamond$ (eventually). The formula $\Box P$ (henceforth $P$) means “$P$ is true for all states in the computation” ($P$ is true now and will remain true forever). The formula $\Diamond P$ (eventually $P$) is interpreted as “there is some state in the computation in which $P$ is true” ($P$ is true now or will become true). The modalities $\Box$ and $\Diamond$ are duals, that is,

$$\Box P \equiv \neg \Diamond \neg P.$$  

When we say that a temporal formula is true for a program, we mean that it is true for all computations of that program.

Temporal operators can be used to express both safety and liveness properties. For example, program termination, a liveness property, can be expressed by the formula

$$P \supset \Diamond \text{ after } P$$

where $P$ and after $P$ are assertions that are true of states in which control is (respectively) at the beginning or end of the program. An example of a safety property is an inductive assertion, that is, an assertion that will remain true if it ever becomes true. The following formula states that $I$ is an inductive assertion:

$$\Box (I \supset \Diamond I).$$

Combinations of the two modalities are also useful. For example, the formula $\Diamond P$ (infinitely often $P$) implies that there are an infinite number of future states for which $P$ is true. (To understand this interpretation, note that $\Diamond P$ implies that $P$ will be true in some future state. The formula $\Box P$ states that this will always be true. In particular, if $P$ ever becomes false, it is guaranteed to become true again at some later time, and this means that it must be true an infinite number of times.) The $\Box$ operator is especially useful for stating recurring properties of a program, for example,

$$\Diamond (\text{the buffer is not full}).$$

We will also use the customary universal and existential quantifiers. Thus, the assertion

$$\exists n(\Diamond (x = 2 \times n))$$

states that $x$ will eventually become even. Note that $n$ in this formula is not a program variable; we will not use formulas in which quantified variable names are also used as program variables in the program under consideration. This should make the meanings of the formulas clear. For another example, consider the assertion

$$\forall i(x = i \supset \Diamond x \geq i).$$

This assertion states that $x$ only takes on values that are at least as large as its initial value, regardless of what that initial value was.

B. Histories

Our proofs use history variables to record the sequence of messages that are the input and output of the modules of the system. History variables have frequently been used in reasoning about communication systems [6], [9], [11], [13], [14].

The initial value of a history variable is the empty sequence, and the only operation allowed is appending a new value.

Suppose $A$ and $B$ are history variables. We write $A \leq B$, to denote that $A$ is an initial subsequence of $B$. This means that $|A| \leq |B|$, and the two sequences are identical in their first $|A|$ elements, where $|A|$ denotes the length of history $A$. If $X$ is a history variable, the following assertion is true for any program containing $X$:

$$\forall A(\Diamond (X \supset A \leq X)).$$

This assertion states that if there is some point at which $X$ has the value $A$, then at all subsequent times $A$ is an initial subsequence of $X$. This follows from the fact that the only operation on a history variable is appending a new value.

We now introduce some notation for describing histories. Let $A$ and $B$ be arbitrary history sequences. If $A$ has elements
Verifying safety properties from code is a well-understood task. To show that an assertion is invariant, one must show that it is true initially, and that it is preserved by each action of the module under consideration (because all of our invariants involve only local and private variables, there is no need to consider interference from other modules). Thus, when we prove that $P$ is invariant, we have proved the temporal assertion

$$\text{Init} \supset \forall Q.$$

where $\text{Init}$ is an assertion that describes the initial state of the program. Proving pre- and postassertions of operations is essentially verifying the partial correctness of the operation’s code. Proofs of these safety properties are discussed further in Owicki [13].

Liveness properties of modules (commitments and live assertions) are proved from code using axioms and inference rules about the liveness properties of program statements, expressed in temporal logic. A summary of this approach is given in Section III-D, and detailed rules are discussed by Halpern, Owicki, and Lamport [7], [15]. Note that when we say that $P$ is a commitment, we mean that

$$\text{Init} \supset \forall Q.$$

just as with invariants. However, commitments are more complex temporal formulas than invariants. For example, commitments often have the form

$$P \supset \forall Q.$$

To state that this assertion is a commitment means that in any computation starting in a legitimate initial state, whenever $P$ becomes true, $Q$ will be true at the same time or later.

Compound modules are verified by showing that the invariants and commitments of the module are implied by the invariants and commitments of its components. At this stage, there is no need to consider the code. This approach lessens the level of detail which must be dealt with at each step. It has the further advantage that the system proof remains valid if any component is replaced by a different implementation that meets the same specifications.

### III. STENNING’S DATA TRANSFER PROTOCOL

To illustrate the application of these program verification techniques to communication protocols we will discuss a simplified version of a data transfer protocol presented by Stenning [19]. (The original version is discussed by Halpern [7].) The protocol is required to deliver all input messages in the order in which they are presented. Stenning verified the safety properties of the algorithm, using a nonmodular proof technique. He did not consider liveness.

Fig. 1 contains the code for the simplified Stenning protocol to be considered, and Fig. 2 is a diagram illustrating the network structure. The protocol is composed of three processes: a transmitter, a receiver, and a communication medium. The transmitter takes as input an unbounded sequence of
HAILPERN AND OWICKI: MODULAR VERIFICATION OF PROTOCOLS

messages \( X \) from source \( X \). It sends them to the receiver, via the communication medium \( mtr \). The receiver outputs messages to sink \( Y \) (the associated output sequence is denoted \( Y \)) and acknowledges receipt via the communication medium \( mrt \). Complications arise because the communication media are unreliable. Messages can be lost, duplicated, or reordered.

(It is assumed that message corruption, if it can occur, is detected by a lower level checksum mechanism, and that corrupted messages are discarded.)

The protocol must ensure that the messages are ultimately delivered correctly in spite of this unreliability. This is accomplished by attaching a sequence number to the messages sent by the transmitter and the acknowledgments sent by the receiver. The transmitter sends each message repeatedly until it receives an acknowledgment of that message, using a timeout mechanism to trigger the retransmission. The first time the receiver gets a message with a given sequence number, it records the message in the output stream. It also sends the transmitter an acknowledgment for every message it receives.

The names of the history variables used in the proof are indicated in the network diagram in Fig. 2. As already mentioned, \( X \) is the input history of the transmitter (and the entire system), and \( Y \) is the output of the receiver (and the system). The input and output histories of the message medium are \( \alpha \) and \( \beta \), respectively, while the input and output histories of the acknowledgment medium are \( \gamma \) and \( \delta \). We denote the unbounded sequence of items to be transmitted by

\[
D = d_1, d_2, \ldots.
\]

These are the values obtained from \( X \), and they appear in the input history \( X \). Note that \( D \) is a global constant sequence, which may be referenced in the proof of any module. On the other hand, \( X \) is a local variable of the transmitter module. A message containing sequence number \( i \) and item \( d_i \) is denoted by \( M_i \), that is,

\[
M_i = [i, d_i].
\]

This is the form of messages in \( \alpha \) and \( \beta \). An acknowledgment for message \( i \), the pair \([i, \text{ "ack"}]\) is denoted by \( A_i \); messages in \( \gamma \) and \( \delta \) have this form. Note that \( d_i, M_i \), and \( A_i \) are constants, whose values do not change throughout program execution.

The remainder of this section presents the proof of Stenning's protocol. We first give the specifications of the communication medium (Section III-A), using temporal logic to make precise assertions about the requirements it must satisfy. We next discuss safety for the simplified protocol, giving the invariants for the transmitter and receiver (Section III-B), and then using them to prove system invariants (Section III-C). The proof of liveness properties in Sections III-D and III-E follows the same pattern.

A. Communication Medium

The communication medium used by the protocol is not defined by program code; it is essentially a black box about which we have limited information. In fact, what we know about the medium is its specifications. (These specifications could be verified by examining the code of lower level components of the system, just as the specifications of the transmitter and receiver can be verified from their code.)
Recall that a module specification involves three kinds of information: invariants, commitments, and service specifications. Because the communication medium we are considering is an unreliable one, it has a very weak invariant: nothing comes out that was not put in,

\[ m \in \beta \supset m \in \alpha \quad (mtr1) \]

\[ m \in \delta \supset m \in \gamma. \quad (mrt1) \]

Note that \( m \) is a free variable, not a program variable. We observe the usual convention that free variables are universally quantified. Thus, \( mtr1 \) states that every message in \( \beta \) is also a message in \( \alpha \). These safety assertions describe a medium which may lose, duplicate, and reorder messages: the only assumption is that it does not create spurious ones.

The invariants above would be satisfied by a medium that never delivered any messages, and in that case no output would ever appear. The medium assumed by Stenning's protocol has two independent commitments that guarantee that some messages ultimately get through. The first is an assertion that if an unbounded number of messages are sent, then messages are infinitely often available to be received.

\[ u(\alpha) \supset \exists m. \text{ExistsM} \quad (mtr2) \]

\[ u(\gamma) \supset \exists m. \text{ExistsM}. \quad (mrt2) \]

In these assertions we used the medium function "ExistsM," which returns the value "true" if at least one message is available. A module function in an assertion is interpreted as true of a state if the function would return "true" when invoked in that state.

The second commitment asserts that if the same message is sent over and over again, it will eventually be delivered (provided that the receiving process keeps accepting messages). This commitment is expressed by the formulas

\[ (uc(\alpha, m) \land u(\beta)) \supset \exists m \in \beta \quad (mtr3) \]

\[ (uc(\gamma, m) \land u(\delta)) \supset \exists m \in \delta. \quad (mtr3) \]

These are about the weakest assumptions we can make and still be able to show that the protocol is able to deliver messages.

Finally, we must specify the operations provided by the medium. In this case there are three: send a message, receive a message, and check to see whether any messages are waiting to be received. The specifications of these services are given below for the medium \( mtr \); the specifications for \( mrt \) are essentially the same.

**send \( (m) \)**

\[
\begin{align*}
\text{pre:} & \quad \alpha = A \\
\text{post:} & \quad \alpha = A(m) \\
\text{live:} & \quad \Diamond \text{after mtr. send}
\end{align*}
\]

**receive \( (\text{var m}) \)**

\[
\begin{align*}
\text{pre:} & \quad \beta = B \\
\text{post:} & \quad \beta = B(m) \\
\text{live:} & \quad \Diamond \text{mtr. ExistsM} \supset \Diamond \text{after mtr. receive}
\end{align*}
\]

Note that a send operation always terminates, and receive terminates if a message is available. The pre- and postassertions of \text{mtr.ExistsM} are both "true," which gives no safety information about the operation. In fact, we will only use \text{mtr.ExistsM} in reasoning about the liveness property that a receive operation terminates.

The timer is another black box, and we could define its properties in a similar way. However, because we will not be doing detailed proofs, it suffices to state that if the timer is set and never canceled, then eventually a timeout notification will be received. Hailpern [7] presents the details omitted here.

### B. Safety: Transmitter and Receiver

Safety specifications of processes are given by invariant assertions about the variables of the process. To verify a process invariant, one shows that it holds initially, and is preserved by each operation of the process. This is a straightforward sequential verification problem. We do not give the details of these verification steps here; we merely state the invariants and explain them informally.

The safety specification of the transmitter consists of two invariants expressed in terms of the abbreviations \( M_i \) and \( A_i \) defined earlier:

\[ \exists \ n(X = (d_i)_{i=1}^n \land (m \in \alpha \supset \exists i \leq n(m = M_i))) \quad (T1) \]

\[ |X| \geq k \supset A_{k-1} \in \delta. \quad (T2) \]

The first invariant states that when \( n \) items have been input to the transmitter, the output to the medium contains only messages that correspond to those \( n \) items with sequence numbers attached. The invariance of \( T1 \) can be proved by noting that it holds initially (when all sequences are empty) and that it is preserved by each operation of the transmitter. We could actually prove a much stronger invariant for this version of the protocol, but we do not need it in the system proof. A formal proof that the transmitter maintains these invariants would include reasoning about the transmitter's changes to \( \alpha \) and \( \delta \), using the pre- and postassertions of \text{mtr.send} and \text{mtr.receive} given in the last section.

The invariant \( T2 \) states that the \( k \)th input term is not read until after the acknowledgment for the \((k - 1)\)st message has been received. This is obvious from the transmitter code.

The receiver has two invariants, which are similar to those of the transmitter:

\[ \forall m(m \in \beta \supset \exists (m = M_i)) \supset \]

\[ (\exists n(Y \geq (d_i)_{i=1}^n) \land d_k \in Y \supset M_k \in \beta) \quad (R1) \]

\[ A_i \in \gamma \supset (M_i \in \beta \land |Y| \geq i). \quad (R2) \]

Roughly speaking, the invariant \( R1 \) states that the receiver output \( Y \) will be legitimate if its input \( \beta \) is legitimate. More
precisely, if \( \beta \) contains only messages of the form \([i, d_i]\), then \( Y \) is a sequence of data items \( \{d_i\}_{i=1}^n \), and each datum in \( Y \) corresponds to a message that appears in \( \beta \). To see that the receiver satisfies this invariant, observe that it will add the \( i \)th element to \( Y \) only after it receives a message with sequence number \( i \), and the value that is appended to \( Y \) is the one contained in the message. By assumption each message in \( \beta \) has the form \([i, d_i]\); therefore, the \( i \)th element of \( Y \) must be \( d_i \).

The second invariant states that if acknowledgment \( a \) is in the output history \( \gamma \), message \( i \) is in the input history \( \beta \), and the associated datum \( d_i \) is in \( Y \). This is obvious from the flow of control in the receiver. An acknowledgment is only sent after the corresponding message has been received and its datum appended to \( Y \).

C. Safety: System

The system safety specifications are also given by an invariant assertion:

\[
Y \leq X. \tag{S1}
\]

This assertion states that the output values are an initial sequence of the input values. It does not imply that any output values are ever produced; that requirement is given in the liveness specifications to be discussed later.

We proceed by assuming the invariants for the transmitter, the receiver, and the communications medium, and showing that the system invariant must follow. As a first step, we note that the hypothesis of the receiver assertion \( R1 \)

\[
\forall m (m \in \beta \Rightarrow \exists i(m = M_i))
\]

follows immediately from the safety properties of the transmitter and medium. Because the transmitter only puts legitimate messages into the medium \( T1 \), and any message that comes out of the medium must have been put in by the transmitter \( mtr1 \), the receiver can only obtain legitimate messages.

Now let

\[
n = \max \{i : M_i \in \beta\}.
\]

Because the hypothesis of \( R1 \) is satisfied, we know that the conclusion of \( R1 \) holds, namely

\[
Y \leq \{d_i\}_{i=1}^n.
\]

But by \( mtr1 \),

\[
M_n \in \beta \Rightarrow M_n \in \alpha
\]

and \( T1 \) implies that if \( M_n \in \alpha \), then \( |X| \geq n \) and hence

\[
M_n \in \alpha \Rightarrow \{d_i\}_{i=1}^n \leq X.
\]

Thus, we can conclude

\[
Y \leq \{d_i\}_{i=1}^n \leq X
\]

which implies the system safety assertion \( S1 \).

D. Liveness: Transmitter and Receiver

The liveness properties of the transmitter and receiver are given by commitments. Verifying that a process satisfies its liveness specifications requires reasoning based on assumptions about the liveness properties of program statements. Formal rules for proving liveness properties from program code are given by Owicki and Lamport [15]. Here we merely state the specifications, and give informal arguments that they are satisfied by the process code. More detailed proofs are given by Hailpern [7].

Our assumption is that processes execute fairly, that is, each process makes progress unless it is blocked. More precisely, let \( s \) be an unblockable action in the program. At \( s \) is the assertion that \( s \) is ready to be executed, and after \( s \) is the assertion that control has finished \( s \). Our basic liveness assumption can be expressed in temporal logic by

\[
at s \supset \diamond after s.
\]

In the protocol system we are discussing, a process can only become blocked while trying to execute a receive operation on a communication medium which has nothing available to be received. The liveness specification for the receive operation \( mtr.receive \) states that receive will terminate if a message is available, that is,

\[
(at mtr.receive \land \Diamond mtr.Exists.M) \supset \diamond after mtr.receive.
\]

Starting from these assumptions about program actions, one can derive rules for proving liveness properties of larger program statements. For example, consider a program statement of the form

\[
\text{loop } S \text{ end loop}
\]

where \( S \) is a statement that does not contain any loops or actions that could be blocked. For such a statement, one can prove

\[
at S \supset \diamond \text{ at } S
\]

that is, control will infinitely often be at the beginning of \( S \). (This is exactly the form of the loop in the transmitter program.) On the other hand, consider the receiver program. Here again we have a loop, but its body \( S' \) contains the statement \( mtr.receive \), which could be blocked. For this loop we can prove

\[
(at S' \land \Diamond mtr.Exists.M) \supset \diamond \text{ at } S'.
\]

The assertion \( \Diamond mtr.Exists.M \) guarantees that whenever \( mtr.receive \) is started, a message will eventually be available so that it can finish execution. Thus, the receive cannot be permanently blocked, and the loop body is executed infinitely often.

So far, we have only talked about liveness properties that involve making progress in the program. More general liveness properties include the effect of program actions on the program variables. For example, from the pre- and postassertion
of send, plus the fact that send can never block, we can conclude

\[(\text{at mtr. send} \land |\alpha| = k) \supset \diamond (\text{after mtr. send} \land |\alpha| = k + 1).\]

For the transmitter, in which mtr.send is embedded in a loop whose body is executed infinitely often, we can conclude

\[\forall (|\alpha| = k \supset \diamond (|\alpha| = k + 1))\]

which implies \(u(\alpha)\).

Now let us consider the liveness specifications of the transmitter. They consist of three commitments. First, the transmitter output history \(\alpha\) grows without bound:

\[u(\alpha).\]  \hfill (T3)

This commitment is independent of any assumptions about the environment. To see that it is satisfied, we note that the transmitter code is a repeating loop which can never be blocked: the only operation that could cause blocking is "receive," and "receive" is only performed when an acknowledgment is provided that the environment keeps making acknowledgments available in mtr. This follows from the absence of blocking, the timeout mechanism guarantees that a message is sent out at least once every timer interval.

The second transmitter commitment is

\[\Box(mtr. \text{ExistsM}) \supset u(\delta).\]  \hfill (T4)

This states that the transmitter will increase the size of \(\delta\) provided that the environment keeps making acknowledgments available in mtr. This follows from the absence of blocking, and the fact that the transmitter will accept an acknowledgment each time around its loop (if one is available).

The third commitment is a promise to start sending the next data item as soon as the current one has been acknowledged:

\[\forall i(A_i \in \delta \supset |X| \geq i) \supset \forall j(A_j \in \delta \supset (uc(\alpha, M_{j+1}) \lor \diamond (A_{j+1} \in \delta))).\]  \hfill (T5)

The hypothesis of this commitment is an assertion that the rest of the system must satisfy: an acknowledgment for message \(i\) is not received before the transmitter has started to work on message \(i\). Under that assumption, once the transmitter receives acknowledgment \(j\), it starts to send message \(j + 1\), and it will send that message an unbounded number of times unless it eventually receives acknowledgment \(j + 1\).

Next we consider the liveness specifications of the receiver. Again we have three commitments, and they are quite similar to the commitments of the transmitter. First, the receiver will cause \(\beta\) and \(\gamma\) to grow unboundedly as long as it is able to receive messages from mtr.

\[\Box mtr. \text{ExistsM} \supset u(\gamma)\]  \hfill (R3)

\[\Box mtr. \text{ExistsM} \supset u(\beta)\]  \hfill (R4)

The receiver code satisfies these assertions because the repeated availability of messages implies that the receiver cannot be blocked at its receive operation. Therefore, it repeatedly executes its loop body, and each time it increases the length of \(\beta\) and \(\gamma\). Note that the transmitter commitment \(T3\), which corresponds to \(R3\), did not require an assumption about the rest of the system in order to guarantee that the size of \(\alpha\) keeps growing. This difference between \(T3\) and \(R3\) comes from the fact that the transmitter uses a timeout mechanism and the receiver does not.

The receiver's third commitment is to acknowledge each message it receives:

\[
(\forall i(M_i \in \beta \supset |Y| \geq i - 1) \land u(\beta)) \supset \\
\forall i(M_i \in \beta \supset (\forall (|Y| \geq i) \\
\land (uc(\gamma, A_i) \lor \diamond (M_{i+1} \in \beta))))].
\]  \hfill (R5)

This commitment is analogous to transmitter commitment \(T5\). Assuming that message \(i\) does not arrive until the receiver has processed message \(i - 1\), and that \(\beta\) grows unboundedly, the receiver will acknowledge each message it receives until the next one arrives, and will add \(d_k\) to the output sequence \(Y\). (It is necessary to assume \(u(\beta)\) because the receiver can block if messages do not arrive; such an assumption is unnecessary for the transmitter, because it can never block.)

E. System Liveness

The system liveness property we ultimately want to prove is that each message is eventually output. Because the safety property tells us that any output produced is an initial segment of the input sequence, all we need to establish is that the output stream gets arbitrarily long, that is,

\[u(Y)\]  \hfill (S2)

Our first step is to prove that all of the medium history variables grow unboundedly. This follows from commitments of the processes and media:

\[u(\alpha)\]  \hfill (T3)

\[u(\alpha) \supset \Box mtr. \text{ExistsM}\]  \hfill (mtr2)

\[\Box mtr. \text{ExistsM} \supset u(\gamma)\]  \hfill (R3)

\[\Box mtr. \text{ExistsM} \supset u(\beta)\]  \hfill (R4)

\[u(\gamma) \supset \Box mtr. \text{ExistsM}\]  \hfill (mtr2)

\[\Box mtr. \text{ExistsM} \supset u(\delta)\]  \hfill (T4)

In combination, these assertions imply that all of the history sequences grow without bound, that is,

\[u(\alpha) \land u(\beta) \land u(\gamma) \land u(\delta)\]

and that input is infinitely often available for mtr.receive and mtr.send.

We now proceed to prove S2, using induction on the length of \(Y\). The induction step is to show that if \(Y\) contains \(k\) messages at some point, then it will eventually contain \(k + \ldots\)
The first step in the proof is to establish the hypotheses of assertions T5 and R5, which state that messages and acknowledgments do not arrive before the recipient is ready to handle them. This is actually a safety property of the system; it often turns out that liveness proofs require additional safety properties. It can be proved easily from the safety specifications of the modules:

\[\{ Y \mid \alpha \} \supseteq \{ Y \mid \beta \} \]

This implies both hypotheses, that is,

\[A_j \in \delta \supseteq |X| \geq j\]  
\[M_j \in \beta \supseteq |Y| \geq j - 1.\]

We now know that the conclusions of T5 and R5 hold, so we can reason with the simpler forms

\[\forall (A_j \in \delta \supseteq (uc(\alpha, M_{j+1}) \lor \diamond (A_{j+1} \in \delta)))\]  
\[\forall (M_j \in \beta \supseteq (\diamond (|Y| \geq j)))\]

\[\land (uc(\gamma, A_j) \lor \diamond (M_{j+1} \in \beta))).\]

Now let us prove the induction step. Suppose that at some point \(|Y| = k\). Then, applying R5', either

\[uc(\gamma, A_k), \quad \text{or} \]

\[\diamond M_{k+1} \in \beta. \quad \text{(2)}\]

Case 1 implies \(\diamond A_k \in \delta\) (from mrt3), and using T5' this in turn implies

\[uc(\alpha, M_{k+1}), \quad \text{or} \]

\[\diamond A_{k+1} \in \delta. \quad \text{(1b)}\]

Now case 1a implies \(\diamond M_{k+1} \in \beta\) (using mrt3), so case 1a reduces to case 2. But case 2 implies \(\diamond |Y| \geq k + 1\) (using R5'). Finally, the system safety relations proved above show that case 1b implies \(\diamond |Y| \geq k + 1\). This completes the proof of the system liveness property.

IV. THE ALTERNATING BIT PROTOCOL

In the field of network protocols, the alternating bit protocol is a classic. It was first published by Bartlett, Scantlebury, and Wilkinson [1] in 1969 in response to a paper by Lynch [12]. Lynch claimed that at least two control bits were necessary to send messages over transmission lines that cause errors. (Here, the term "error" implies that the communication medium can corrupt the contents of a message.) The alternating bit protocol, as shown in the finite-state machine of Fig. 3, requires only one bit of control information to guarantee reliability.

The original protocol consists of two processes and a communication medium; both processes can send and receive data from outside users. We take a slightly different view of the protocol by restricting the services that the two processes provide and by including a second medium. Only process A will receive data from users, and only process B will send data to users. These changes do not significantly modify the problem, but they make the proof easier to understand. Process A reads data from an external unbounded source of data \(X\). The history of the data that the process has read is called \(X\). A sequence bit and a datum are combined into a message, which is sent to process B by way of communication medium \(ma\). Process B receives the messages from \(ma\), strips off the sequence bit, and outputs the data to the unbounded sink \(Y\). The history of the data that the process has output is denoted \(Y\). Acknowledgments, consisting of a sequence bit and the datum "ack," are sent back to process A by way of medium \(mb\). The histories \(\alpha, \beta, \gamma, \delta\) record the messages sent to and from the media. Fig. 5 shows the system diagram. The media are modeled as single-element buffers that can change a sequence number to the constant "error." Such a change represents a corruption of the datum. (This model of corruption is reasonable if we assume the existence of a lower level mechanism that detects corrupted messages by using a checksum. The corrupted messages can be reported in the manner described above.) Our goal is to show that the protocol delivers messages in the correct order, in spite of the possibility of corruption by the medium. The code for processes A and B is presented in Fig. 4.

A. The Communication Medium

The communication media assumed by the alternating bit protocol are somewhat more reliable than those discussed for Stenning's protocol. Messages can be corrupted (in a detectable way), but they cannot be lost, duplicated, or re-
A: process
  var
  WaitingForAck, LastSent: integer { modulus = 2 }
  data, ack: item
  ackno: (0, 1, error)
  a, X: private history
begin
  | initialize |
  WaitingForAck := 1
  LastSent := 0
loop
  | has the current message been acknowledged? |
  if (LastSent = WaitingForAck) then
    | read in new data |
    LastSent := LastSent ⊕ 1
    (X), read(data)
  fi
  | send new message |
  mab. send( [LastSent, data] )
  | wait for acknowledgment |
  mba. receive( [ackno, ack] )
  | is this an acknowledgment for the current message? |
  if (ackno = WaitingForAck) then
    WaitingForAck := WaitingForAck ⊕ 1
  fi
end loop
end process

B: process
  var
  NextRequired: integer { modulus = 2 }
  messno: (0, 1, error)
  info: item
  β, γ: private history
begin
  | initialize |
  NextRequired := 1
loop
  | wait for next message |
  mab. receive( [messno, info] )
  | is this a new message? |
  if (messno = NextRequired) then
    | process new message |
    (Y), write(info)
    NextRequired := NextRequired ⊕ 1
  fi
  | send acknowledgment |
  mba. send( [NextRequired ⊕ 1, "ack"] )
end loop
end process

Fig. 4. Alternating bit protocol. (a) Process A. (b) Process B.

Fig. 5. System diagram for the alternating bit protocol.

The first invariant expresses this assumption about data transfer:

\[(β_i \neq α_i) \lor \text{corrupted}(β_i)\]

\[(δ_i \neq γ_i) \lor \text{corrupted}(δ_i),\]

where \text{corrupted}(m) is true if the sequence bit of message \(m\) is "error." The corresponding invariants for the communication media for Stenning's protocol \((mtr1\) and \(mrt1)\) merely asserted that any message that appeared in the output history must also appear in the input history.

The media for the alternating bit protocol have a fixed buffering capacity of one message. This means that a medium must be either empty or full, and that the input history can be at most one element longer than the output history. These facts are expressed by the next two invariants:

\[\text{mab. empty} \equiv \neg \text{mab. full} \quad (mab2)\]

\[\text{mba. empty} \equiv \neg \text{mba. full}. \quad (mba2)\]

If \text{mab. empty} then \(|α| = |β|\) else \(|α| = |β| + 1 \quad (mab3)\]

If \text{mba. empty} then \(|γ| = |δ|\) else \(|γ| = |δ| + 1. \quad (mba3)\]

Because the media for the Stenning protocol had an unbounded capacity, there were no invariants analogous to these.

There is one liveness commitment for the alternating bit media. If message \(m\) is sent an unbounded number of times, and an unbounded number of receives are performed, then message \(m\) is received correctly an unbounded number of times. This is expressed by the commitment

\[uc(α, m) \land u(β) \supset uc(β, m) \quad (mab4)\]

\[uc(γ, m) \land u(δ) \supset uc(δ, m). \quad (mba4)\]

Note that, this is the same as commitments \(mtr3\) and \(mrt3).\ The Stenning media had another liveness commitment, which is not needed here.

Each medium provides two services, \text{send} and \text{receive}, and two auxiliary functions, \text{empty} and \text{full}. The services for \text{mab} are shown below (the services for \text{mba} are similar). The pre- and postassertions are the same as for the Stenning medium, but the live assertions are somewhat different. In particular, the send operation can be blocked if the medium is full.

send \((m)\)

live: \(\Box (\text{mab. empty}) \supset \Box (\text{after mab. send})\)

receive (var \(m)\)

live: \(\Box (\text{mab. full}) \supset \Box (\text{after mab. receive}).\)

B. Safety: Process A:

As before, we denote the \(i\)th data value to be transmitted by \(d_i\). The messages in history \(α\) contain a data value and a single bit of sequencing information. The message for the
ith datum is \( M_i \), where
\[
M_i = [i \mod 2, d_i].
\]

The first invariant of process \( A \) relates the histories \( X \) and \( \alpha \):
\[
\exists n(X = \langle d_i \rangle_{i=1}^{n} \land \alpha \in \langle M_i \rangle_{i=1}^{n} \langle M_i \rangle^+. )
\] (A1)

The superscripts in invariant A1 are derived from those used in regular expressions. In other words, when \( n \) items have been input to process \( A \), the output to \( mab \) is a sequence of repeated messages: one or more copies of \( M_1 \), then one or more copies of \( M_2 \), and so on, ending with zero or more copies of \( M_n \). The last term is \( M_n^+ \), rather than \( M_n^* \), because after the operation \( \bar{X} \) read, the \( n \)th data item has been read into \( X \), but message \( M_n \) has not yet been sent. The invariance of A1 is easy to see, because process \( A \) repeatedly reads a data item and then transmits the corresponding message until it receives an acknowledgement. Although this invariant is considerably stronger than \( T1 \), the transmitter in the simplified form of Stenning’s protocol actually satisfies the invariant A1. However, Stenning’s protocol was designed to deal with a medium in which messages could be reordered, so the stronger invariant was not necessary.

The second invariant of process \( A \) relates the sizes of \( \alpha \) and \( \delta \):
\[
| \delta | \leq | \alpha | \leq | \delta | + 1.
\] (A2)

This follows from the fact that process \( A \) repeatedly executes a loop in which it sends a message via \( mab \) and then accepts a message from \( mba \).

The third invariant establishes the correspondence between acknowledgments in \( \delta \) and messages in \( \gamma \). In Stenning’s protocol this correspondence was obvious, because both messages and acknowledgments contained a full sequence number. In the alternating bit protocol, they contain only a single bit of sequencing information. However, the full sequence number can be determined by examining the bit fields of the preceding messages in the history: a change in parity between successive elements signals a change in sequence numbers.

We define \( \#(\alpha, i) \) to be the number of changes in parity in \( \alpha \) up to and including element \( i \); thus, it is the sequence number of the \( i \)th element in the sequence. More precisely,
\[
\#(\alpha, 0) = 0
\]
\[
\#(\alpha, 1) = 0 \quad \text{if } \alpha_1. \text{ bit} = 0
\]
\[
\#(\alpha, 1) = 1 \quad \text{if } \alpha_1. \text{ bit} = 1
\]
\[
\#(\alpha, i) = \#(\alpha, i - 1) \quad \text{if } i > 1 \land \alpha_i. \text{ bit} = \alpha_{i-1}. \text{ bit}
\]
\[
\#(\alpha, i) = \#(\alpha, i - 1) + 1 \quad \text{if } i > 1 \land \alpha_i. \text{ bit} \neq \alpha_{i-1}. \text{ bit}
\]

Note that a state with \( \#(\alpha, 1) = 0 \) cannot occur in this system, because A1 states that if \( | \alpha | \geq 1 \), then \( \alpha_1. \text{ bit} = 1 \). We will denote the last sequence number in the history by \( \#(\alpha) \), where
\[
\#(\alpha) = \#(\alpha, | \alpha |).
\]

Determining the sequence numbers of acknowledgments in \( \delta \) is a bit more complicated, because \( \delta \) may contain errors. Elements of \( \delta \) have the form
\[
[\text{bit, "ack"}]
\]

where “bit” is an element of \( \{0, 1, \text{error}\} \). In order to define \# on \( \delta \), we use a projection \( \hat{\delta} \) that contains only the non-corrupted elements of \( \delta \):
\[
\hat{\delta} = \text{project } (\delta, \text{bit} \neq \text{error}).
\]

A projection of a history on a Boolean expression creates a new history that contains only those elements for which the Boolean expression is true. The order of the elements of the original sequence is preserved in the projected sequence. The new sequence \( \hat{\delta} \) has only the acknowledgments with bit equal to 0 or 1.

We can define \( \#(\hat{\delta}, i) \) in the same way as \( \#(\alpha, i) \). Now let the projection \( \pi \) take \( \delta_i \) to \( \hat{\delta}_{\pi(i)} \). We can extend \# to \( \delta \) as follows:
\[
\#(\delta, i) = \begin{cases} 
\#(\hat{\delta}, \pi(i)), & \text{if } \sim \text{ corrupted } (\delta_i) \\
\#(\delta, i - 1), & \text{if } \text{corrupted } (\delta_i),
\end{cases}
\]

where \( \#(\delta, 0) = 0 \).

We are now ready to relate the parity changes of \( \alpha \) to those of \( \delta \). The third invariant for process \( A \) states that the \( i \)th message is not sent until the \( (i - 1) \)st acknowledgment has been received and the \( i \)th item has been read from \( X \):
\[
\#(\delta) \leq \#(\alpha) \leq |X| \leq \#(\delta) + 1.
\] (A3)

The invariance of this assertion follows easily from the code: it is only after a change in parity is detected in \( \delta \) that a new element is read from \( X \) and then sent to \( \alpha \).

C. Safety: Process B

The first three invariants for process \( B \) are much the same as the invariants for process \( A \). We can define \( \#(\gamma, i) \) in the same way that we defined \( \#(\alpha, i) \), because neither \( \alpha \) nor \( \gamma \) contains any errors. Because both \( \delta \) and \( \beta \) can contain errors, we define \( \#(\beta, i) \) in the same way as \( \#(\delta, i) \). We also define \( A_i \) as the pair \( [i \mod 2, "ack"] \). The first three invariants are then
\[
\exists n(X = n \land \gamma = \langle A_0 \rangle^n \langle A_i \rangle_{i=1}^n \langle A_n \rangle^+)
\] (B1)
\[
|\gamma| \leq |\beta| \leq |\gamma| + 1
\] (B2)
\[
\#(\gamma) \leq |Y| \leq \#(\beta) \leq \#(\gamma) + 1.
\] (B3)

The fourth invariant states that process \( B \)’s output \( Y \) will
be legitimate if its input $\beta$ is legitimate:

$$
(\sim \text{corrupted}(\beta|\beta|) \supset \beta|\beta| = M_{#(\beta)})
$$

$$
\supset Y \leq (d|_{i=1}). \quad (B4)
$$

The hypothesis of $B4$ (which will be proved as a system safety property) implies that the uncorrupted messages in $\beta$ are legitimate messages (they contain a datum from $X$ and the appropriate sequence bit) and they are preceded by the appropriate number of parity changes. The $i$th element is added to $Y$ only after process $B$ receives a message with derived sequence number $i$, and the value that is appended to $Y$ is the one contained in the message.

D. Safety: System

The system invariant for the alternating bit protocol is the same as for Stenning's protocol:

$$
Y \leq X. \quad (S1)
$$

Its proof requires three further system invariants, which can be derived from the conjunction of the module invariants.

The first invariant relates the sequence numbers of elements in $\alpha$ and $\beta$.

$$
#(\alpha, |\beta|) - #(\beta) \geq 0 \lor
\forall i((\beta_i = \alpha_i) \supset (#(\alpha, i) - #(\beta, i) \text{ is even})). \quad (S2)
$$

Proof of $S2$: This invariant is a consequence of $mab1$ and the definition of $\#$. $mab1$ states

$$
\forall j |\beta| (\beta_j = \alpha_j \lor \text{corrupted}(\beta_j)).
$$

Using induction on the definition of $\#$, we see that corrupting an element in a sequence cannot result in a larger value for $\#$. This establishes the first clause of $S2$. The second clause follows from the fact that $\#$ counts the number of changes of parity. The sequences start out with the same parity (the parity of the empty sequence is 0), and end with the same parity (because $\alpha_l = \beta_l$). Thus, either $#(\alpha, l)$ and $#(\beta, l)$ are both odd or they are both even. In either case their difference is even.

Because $#(\gamma)$ and $#(\delta)$ are related in the same way as $#(\alpha)$ and $#(\beta)$, we have a similar invariant relating their values:

$$
#(\gamma, |\delta|) - #(\delta) \geq 0 \lor
\forall i((\delta_i = \gamma_i) \supset (#(\gamma, i) - #(\delta, i) \text{ is even})). \quad (S3)
$$

Finally, we add an invariant that relates the $#$ values of all the histories in the system.

$$
(|X| - #(\alpha)) + (#(\alpha) - #(\beta)) + (#(\beta) - |Y|)
$$

$$
+ (|Y| - #(\gamma)) + (#(\gamma) - #(\delta)) + (#(\delta) + 1 - |X|)
$$

$$
= 1 \land \text{each term in the sum is between 0 and 1}. \quad (S4)
$$

This invariant implies that there is at most one point in the system where there is a message to be processed.

Proof of $S4$: Algebraic simplification establishes that the sum in $S4$ is 1. The fact that each term is nonnegative follows from the process invariants and the system invariants just proved.

$$
(|X| - #(\alpha)) \geq 0 \land (#(\delta) + 1 - |X|) > 0 \quad (A3)
$$

$$
(#(\beta) - |Y|) \geq 0 \land (|Y| - #(\gamma)) \geq 0 \quad (B3)
$$

$$
(#(\alpha) - #(\beta)) \geq 0 \land ( #(\gamma) - #(\delta)) \geq 0. \quad (S2, S3)
$$

Because each difference is nonnegative, and their sum is 1, each term in the sum must be no greater than 1.

We are now ready to prove the system invariant $S1$, that is,

$$
\sim \text{corrupted}(\beta|\beta|) \supset \beta|\beta| = M_{#(\beta)}).
$$

Let $n = |\beta|$. We know from $mab1$ that

$$
\sim \text{corrupted}(\beta_n) \supset \beta_n = \alpha_n
$$

and from $A1$ that

$$
a_n = M_{#(\alpha, n)}.
$$

So the hypothesis of $B4$ will be proved once we have shown

$$
\sim \text{corrupted}(\beta_n) \supset (#(\alpha, n) = #(\beta, n)),
$$

that is, that $\alpha$ and $\beta$ assign corresponding sequence numbers for the last element in $\beta$ unless it is corrupted.

Now assume $\sim \text{corrupted}(\beta_n)$. We know from $S4$ that $#(\alpha) - #(\beta) \leq 1$. We can rewrite this difference, using the fact that $#(\beta, n) = #(\beta)$ to obtain the equivalent relation

$$
(#(\alpha) - #(\alpha, n)) + (#(\alpha, n) - #(\beta, n)) \leq 1.
$$

The first term in the sum is nonnegative, given the definition of $#$ and the fact that $mab3$ implies

$$
|\alpha| \geq |\beta| = n.
$$

Thus, the second term must be no greater than 1; it is also nonnegative and even ($S2$). Because the only nonnegative even integer less than 2 is 0, we have the required result that

$$
#(\alpha, n) = #(\beta, n).
$$

This establishes the hypothesis of $B4$, so we can conclude $Y \leq (d|_{i=1})$. The relation $Y \leq X$ then follows from $A1$.

E. Liveness: Processes $A$ and $B$

Now let us consider the liveness specifications of process $A$. They consist of four commitments. First, the process will initially put something into the buffer $mab$, because there is no way it can become blocked before one value is put in.
Init $\Diamond \top {\text{mab. full.}}$

Second, if the process does not become blocked, the output history $\alpha$ grows without bound:

$\Diamond \top \text{mab. empty} \land \Diamond \top \text{mba. full} \supset u(\alpha).

We justify $A5$ by noting that if there is no blocking, then one message is sent on each cycle of the loop.

The third commitment states that whenever there is something in buffer mba, there will eventually be something in buffer mab:

$mab. \text{full} \supset \top \text{mab. full.}

To see that this is the case, note that process $A$ repeatedly executes its loop unless it is blocked because mba is empty or because mab is full. Given the hypothesis that mba is full, the next execution of the loop body cannot become blocked at mba.receive. Thus it either adds an element to mab or becomes blocked because mab is full. In either case, we have $\Diamond \top \text{mab. full.}$

The fourth commitment is a promise to start sending the next data item as soon as the current one has been acknowledged:

$u(\alpha) \supset \forall j (\text{#(\delta)} \supset j \supset (w(a, M_{j+1}) \lor \text{#(\delta)} \geq j + 1)).

The hypothesis of $A7$ implies that the process does not become permanently blocked. Under that assumption, if process $A$ receives acknowledgment $j$, then $A$ starts to send message $j + 1$; it will send that message an unbounded number of times unless it eventually receives acknowledgment $j + 1$.

Next consider the liveness specifications of process $B$. There are three commitments, and they are essentially the same as $A5$-$A7$. (There is no commitment analogous to $A4$, because process $B$ can be blocked before it produces any output.)

$\Diamond \top \text{mab. empty} \land \Diamond \top \text{mab. full} \supset u(\gamma).

$\top \text{mab. full} \supset \Diamond \top \text{mab. full.}

$u(\gamma) \supset \forall j (\text{#(\beta)} \supset j \supset (u(\gamma, A_j) \lor \text{#(\beta)} \geq j + 1))).

F. System Liveness

System invariant $S1$ states that if any output appears in $Y$, it is the same as the input in $X$. The liveness property that we want to prove is that the length of the output history increases without bound:

$u(Y).

Our proof will be structured much like the proof of system liveness for Stenning's protocol. The first step will be to show that the buffer histories grow without bound, that is, that values are repeatedly transmitted between the processes. This will establish the hypotheses of the process commitments $A7$ and $B7$, and system liveness follows in much the same way as before. However, the reason that the buffer histories grow without bound is somewhat different in the two protocols. In Stenning's protocol, the transmitter included a timeout mechanism that repeatedly retransmitted messages that had not been acknowledged. Thus, the transmitter caused $\alpha$ to grow unboundedly, and then the commitments of the communication medium and the receiver guaranteed that $\beta$ would grow unboundedly. The alternating bit protocol does not rely on timeouts. Instead, the transmitter sends a message each time it receives an acknowledgment from the receiver. If the receiver stopped sending acknowledgments, the transmitter would become blocked and stop sending messages, and vice versa. Because the buffers have a bounded capacity, each process could also become blocked if the other stopped removing items from its input buffer. Our proof must establish that the processes cooperate in such a way that neither becomes blocked.

To demonstrate this cooperation, we need the following system invariant, which implies that at most one of the communication buffers can be full at any time:

$\sim \top \text{mab. full} \land \top \text{mab. full}.$

Proof of $S6$: Consider the equation

$(|\alpha| - |\beta|) + (|\beta| - |\gamma|) + (|\gamma| - |\delta|) + (|\delta| + 1 - |\alpha|) = 1.

This is obviously an invariant, because the left-hand side of the equation simplifies to 1. Moreover, each term in the sum is always between 0 and 1; the proof is similar to that of $S4$, using invariants $A2, \text{mab3}, B2,$ and $\text{mab3}$. This means that at any time, exactly one term in the sum is equal to 1 and the rest are equal to 0. Now $\text{mab3}$ and $\text{mab3}$ imply

$\top \text{mab. full} \equiv (|\alpha| - |\beta| = 1)

\top \text{mab. full} \equiv (|\gamma| - |\delta| = 1).

Because at most one of these differences can be positive at a time, at most one buffer can be full at a time.

We are now ready to prove the “no starvation” commitment

$u(\alpha) \land u(\beta).

Proof of $S7$: We will prove $u(\beta); u(\alpha)$ will follow from $\text{mab3}$. We can use $B5$ to establish $u(\beta)$: it is only necessary to show

$\Diamond \top \text{mab. empty} \land \Diamond \top \text{mab. full,}$

or, equivalently,

$\Diamond \sim \top \text{mab. full} \land \Diamond \top \text{mab. full.}$
To show $\bowtie mab.full$, we will show that there is some time at which $mab.full$ is true, and that if it is true at any time $i$ there is a later time $k$ at which it is true again. The fact that $mab.full$ is true at some time comes from the commitment $A_4$. Now suppose $i$ is some time at which $mab.full$ is true. By $B_6$, there is a time $j > i$ such that $mba.full$ is true at time $j$. But $mab.full$ and $mba.full$ cannot be true at the same time ($S_6$), so $j > i$. Now by $A_6$, there is a time $k > j$ such that $mab.full$ is true at time $k$. Thus, for any time $i$ at which $mab.full$ is true, there is a later time $k$ at which it is true again. This implies $\bowtie mab.full$. Because only one buffer at a time can be full, this implies $\bowtie \sim mba.full$, and that completes the proof of $S_7$.

Note that this proof has been carried out quite informally, to avoid introducing a number of temporal logic theorems that are not important in this context. A more formal proof can be constructed using the axioms and theorems presented by Halpern [7, Appendix A].

Having proved that the hypotheses of $A_7$ and $B_7$ hold, we know that the conclusions also hold:

$$\forall i ((\#(\delta) > j \cup (\delta(\alpha, M_{i+1}) \cup/(\#(\delta) > j + 1))) \quad (A_7')$$

$$\forall i ((\#(\delta) > j \cup (\delta(Y) > j) \cup/(\#(\delta) > j + 1))) \quad (B_7')$$

We are now ready to prove the system liveness property $S_S$.

**Proof of $S_5$:** The proof of $S_5$ is similar to the proof of the corresponding property in Stenning’s protocol ($S_2$).

**V. CONCLUSION**

In addition to the protocols presented here, we have proved the correctness of Brinch Hansen’s multiprocessor network [4]. We have found that program verification techniques can be used to prove the safety and liveness of network protocols that handle an unreliable medium. By insisting on modular decomposition and restricting the view of implementation details, we are able to manage the complexity of program proofs. Temporal logic is an important tool, which allows us to state liveness properties in a clear, consistent manner.

**REFERENCES**


Brent T. Halpern (S’80–M’80) was born in Denver, CO, on January 11, 1955. He received the B.S. degree in mathematics from the University of Denver, Denver, in 1976, and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA, in 1978 and 1980, respectively.

He has been employed as a Research Staff Member in the Computer Sciences Department of the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, since 1980, where he is working in the area of office systems. His research interests include concurrent programming, program verification, network protocols, and distributed systems.

Dr. Halpern is a member of the Association for Computing Machinery, Phi Mu Epsilon, and Sigma Xi.

Susan S. Owicki was born in Pittsburgh, PA on June 25, 1947. She received the B.S. degree in mathematics from Michigan State University, East Lansing, in 1968, and the M.S. and Ph.D. degrees in computer science from Cornell University, Ithaca, NY, in 1970 and 1975, respectively.

From 1975 to 1976 she was an Assistant Professor in the Department of Computer Science at Cornell University. Since then she has been a member of the Department of Electrical Engineering at Stanford University, Stanford, CA, where she is currently an Associate Professor. Her research interests in the area of concurrent programming include program verification, programming languages, and programming methodology. She is also interested in medical applications of computers.

Dr. Owicki is a member of the Association for Computing Machinery, Phi Eta Sigma, and Tau Beta Pi. She is an affiliate member of the IEEE Computer Society. From June 1979 to June 1982, she was a member of the Computer Science Board of the National Research Council of the National Academy of Science.