A Simple Protocol Whose Proof Isn't

BRENT HAILPERN, SENIOR MEMBER, IEEE

Abstract—**Aho, Ullman, and Yannakakis have proposed a set of protocols that ensure reliable transmission of data across an error-prone channel. They have obtained lower bounds on the complexity required of the protocols to assure reliability for different classes of errors. They specify these protocols with finite-state machines. Although the protocol machines have only a small number of states, they are nontrivial to prove correct. In this paper we present proofs of one of these protocols using the finite-state-machine approach and the abstract-program approach. We also show that the abstract-program approach gives special insight into the operation of the protocol.**

I. Introduction

HOW complex must a protocol be to guarantee reliable communication? The answer, of course, depends on the communication medium provided, the type of communication desired, and the definition of *reliable*. Aho, Ullman, and Yannakakis [1] have derived lower bounds on the complexity needed to achieve reliable communication across an error-prone channel. They measure complexity by the size of the finite-state machines used to specify the protocol. In particular, they describe a transmitter–receiver system where each machine has only two states, and a protocol that guarantees reliable communication over a medium that can lose messages. In this paper we verify the correctness of their protocol using two different techniques: the finite-state approach and the abstract-program approach.

Section II describes the model on which the protocol is based. Section III presents the finite-state approach to protocol verification. It also includes a proof of the protocol under discussion. Section IV describes the abstract-program approach. We include an introduction to two tools of the approach: history variables (to describe the input–output properties of a protocol) and temporal logic (to provide a notation for showing that the protocol is *live*—for example, that it will continue to send and receive messages as desired). Section V discusses history variables and safety specifications in more detail. In Section VI we prove that the protocol is correct under the assumption that no messages are lost. Section VII discusses temporal logic and liveness properties. In Section VIII we outline the proof that the protocol is correct even if messages can be lost. Finally, we comment on the two verification techniques, as they were applied to this protocol.

II. The Model

The model that Aho, Ullman, and Yannakakis assume for their protocol (henceforth called the AUY protocol) is a pair of finite-state machines that communicate over an error-prone channel. The transmitter's input is an (unbounded) sequence of bits; the desired output of the receiver is the same sequence of bits. The two automata are synchronous; that is, state transitions in the two machines must occur simultaneously. The system contains a central clock: at each tick of the clock, each machine undergoes a state transition and transmits a symbol (one of 0, 1, or λ) to the other machine. The symbol λ represents an empty or lost message. Fig. 1 shows a diagram of the system: finite-state machine A is the transmitter and finite-state machine B is the receiver. The source of bits for the transmitter is the sending user S, and the destination for the bits from the receiver is the receiving user R. A *move* of A (that which occurs at each tick) depends on the current state of A, the symbol transmitted by B during the previous tick, and the current symbol in the input buffer. Based on these three factors, A makes its move, which consists of a change of state, a transmission of a symbol to B, and an optional advance to the next symbol of the input buffer. The movements of B are similar. The model addresses three types of transmission errors:

- deletion error—0 or 1 is sent, λ is received,
- mutation error—0 or 1 is sent, 1 or 0 is received,
- insertion error—λ is sent, 0 or 1 is received.

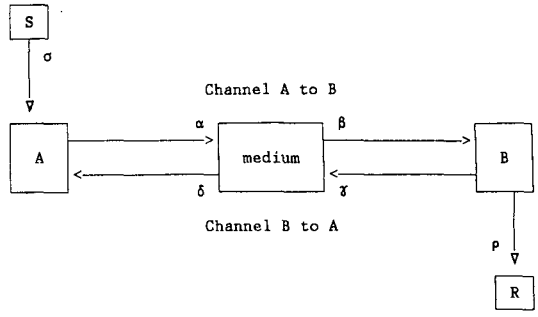


Fig. 1. AUY protocol configuration.

Aho, Ullman, and Yannakakis concentrate on a protocol to handle only deletion errors, and so shall we.

The finite-state machines of the AUY protocol are shown in Fig. 2. The four states are labeled p, q, r, and s. The arcs, representing state transitions, are labeled by the conditions under which the transitions may be taken and by the events associated with the transitions. Machine A, the transmitter, has two input sources: the sending user and machine B. The bits from the sending user are the messages to be sent by the protocol. The bits from B are acknowledgments. These sources are represented on the finite-state diagram by an ordered pair in parentheses, for example (su, ack), where su represents the bit from the sending user and ack represents the acknowledgment bit from B. Machine A has a single output—to machine B. The output bit is indicated on the diagram by being enclosed in angle brackets, $\langle b \rangle$. During a move, machine A can advance the input buffer by one symbol, indicating that the next bit from the sending user is available to be read (on the next clock tick). This action is indicated by a star (*). Therefore, a transition of the transmitter labeled (0, 1) $\langle \lambda \rangle^*$ shows that if the input from the sending user equals 0 and the acknowledgment from the receiver equals 1, then the transmitter can take this transition. Taking this transition causes a λ to be sent to machine B and the input buffer to be advanced.

Similarly the transitions of machine B are labeled $(a)\langle ack, ru \rangle$, where a is the input from A, ack is the acknowledg-

Paper approved by the Editor for Computer Communication of the IEEE Communications Society for publication after presentation at the Trends and Applications Symposium on Advances in Software Technology, Gaithersburg, MD, May 1981. Manuscript received August 20, 1982; revised January 5, 1984.

The author is with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

0090-6778/85/0400-0330$01.00 © 1985 IEEE

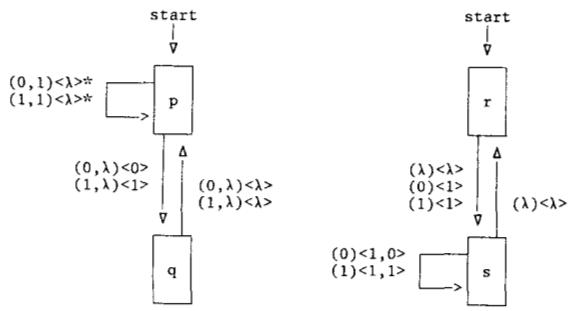


Fig. 2. Finite-state specification.

Step	Current Input	Channel Symbol Seen by A	A State	Channel Symbol Seen by B	B State	Output
(1)	0	λ	p	λ	r	
(2)	0	λ	q	0	s	0
(3)	0	1	p	λ	s	
(4)	1	λ	p	λ	r	
(5)	1	λ	q	λ lost	s	
(6)	1	λ	p	λ	r	
(7)	1	λ	q	1	s	1
(8)	1	1	p	λ	s	
(9)	0	λ	p	λ	r	
(10)	0	λ	q	0	s	0
(11)	0	λ lost	p	λ	s	
(12)	0	λ	q	0	r	
(13)	0	λ lost	p	λ	s	
(14)	0	λ	q	0	r	
(15)	0	λ	p	λ	s	
(16)	1	λ	p	λ	r	
(17)	1	λ	q	1	s	1
(18)	1	1	p	λ	s	
(19)	none	λ	p	λ	r	

Fig. 3. Possible sequence of transitions on 0101, from [1]. Bits that have been lost in transit are indicated by the notation "lost."

ment sent to A, and ru is the bit sent to the receiving user. If a transition does not cause a bit to be sent to the receiving user, then the ru component will be omitted. Fig. 3 [1] shows a possible sequence of transitions on the input 0101.

By what standards is a protocol for such a system correct? Aho, Ullman, and Yannakakis describe a protocol, a pair of automata, as being *robust* under a certain class of errors if it satisfies the following two properties. First, the output sequence is always a prefix of the input sequence. Second, if no transmission errors occur for some fixed length of time, then at least one more output symbol is printed. We show that their protocol is correct in two different ways. We first perform a finite-state proof, which uses reachability analysis to identify deadlocked states. We then construct an abstract-program proof using the Floyd–Hoare technique augmented by temporal logic. In neither case will our definition of *correct* correspond exactly to their notion of *robust*.

III. Finite-State Technique

To verify the correctness of a protocol using the finite-state technique, one forms the system state space as the cross product of the states of the component machines and media. One then does a reachability analysis by starting in the initial state and exercising all legal transitions. The reachability analysis terminates when all legal transitions have been tried from all reachable states. If any reachable state has no exit transitions, then deadlock is possible. If a system state is reachable that is not defined in the component machines, then the protocol is not completely specified for all possible inputs.

This technique does not address the notion of correct data transfer. Bochmann [2] does address correct data transfer by showing that the action sequence (or language) generated by the system machine (finite state machine formed in the reachability analysis) has each "input I" symbol followed by exactly one "output I," except when the last "output I" has not occurred. Gouda [10] also proves correct data transfer by forming a labeled finite reachability graph (with variables). He then treats this graph as a sequential program and proves correct data transfer using conventional program verification techniques.

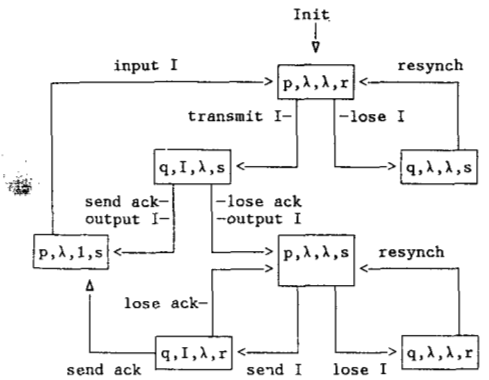


Fig. 4. Reachability analysis of AUY protocol. Note that "I" indicates either 0 or 1. Therefore, this figure is not the full system automata—each "I" transition should be duplicated.

Excellent surveys of the finite-state technique have been written by Bochmann, Danthine, and Sunshine [3], [4], [7], [30].

In the AUY protocol there are two machines, each having two states; there are two channels, each of which can be in one of three states (0, 1, λ). Therefore, there are 36 system states. Fig. 4 shows the results of the reachability analysis for the AUY protocol. A system state is represented by the quadruple (a, msg, ack, b), where a is the transmitter state, b is the receiver state, msg is the message being sent from A to B, and ack is the acknowledgment sent from B to A. In fact, there are no deadlocked or undefined states.

Does this verification of the protocol show that it is robust? The finite-state technique does not address the issue of correct data transfer, which is the first component of the definition of *robust*. Note that data transfer could be verified, if the finite state machines were augmented with variables at each state, but then the proof would be more complicated than a simple reachability analysis.

The second part of the definition can be demonstrated by trying all possible starting positions in the system state space and showing that if no deletions occur, then an output symbol must be produced after a certain fixed number of moves. For the AUY protocol the fixed number is 6.

IV. Abstract-Program Technique

In the abstract-program approach, the protocol is modeled as a parallel program. Correctness proofs follow the Floyd–Hoare style of program verification [8], [14], [20]. Logical assertions attached to the program abstract information from the representation of the state, allowing us to reason about classes of states. Hailpern, Schwartz, and Melliar-Smith have written surveys of abstract-program techniques as applied to network protocols [12], [27].

In our version of the technique, we model the transmitter and receiver as processes; the media are represented by synchronized shared variables (described below). We use a modular approach to verification: we first prove properties of the individual processes, and then we combine these properties into a proof of the entire protocol.

In the abstract-program approach, we are interested in two kinds of properties: safety and liveness. 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; that is, an invariant assertion is true before and after the execution

of each statement in the program. For example, freedom from deadlock is a safety property. We often express safety properties using (auxiliary) history variables that record the interactions of the processes. The term *auxiliary* indicates that the history variables are not actually implemented. Such variables are used only in the proof. Because they are not implemented, they can record histories of unbounded length. History variables have frequently been used in reasoning about communication systems [9], [13], [16], [21], [22].

In our system, we will maintain histories recording the input from the sending user and the output to the receiving user (represented by σ and ρ in Fig. 1). These two histories will allow us to state that the output of the protocol is a prefix of the input to the protocol. In addition, we will use histories to record the bits sent and received by the two processes (represented by α, β, γ, and δ in Fig. 1). Histories are discussed in more detail in Section V.

Liveness properties have the form "good things will happen." Indeed, if we have shown that no bad things can happen, then the form is simply "things will happen." These properties include termination requirements in sequential programs. Some programs, however, are not intended to terminate, for example, operating systems. For such programs, liveness properties assert that some desirable state will reoccur or that some action will eventually be performed.

Liveness properties refer to the future occurrence of a desired state. As such, they are difficult to state in conventional logic, which refers only to the present state. To state liveness properties, we use the notation of temporal logic [18], [24], [25], which provides operators for describing future program states. Section VII discusses temporal logic in more detail.

In translating the AUY protocol from a finite-state machine to an abstract program, we need program constructs to represent the operations of reading a new bit from the sending user, sending a bit to the receiving user, and exchanging bits between the two processes. The first two operations are simple. The command

$$x \leftarrow S.read$$

will input the next value from the sending user, and the command

$$R.write(x)$$

will output x to the receiving user. We assume that *read* and *write* are guaranteed to terminate. The liveness specification of these operations is in Section VII. The *exchange* command is similar to the send and receive operations (! and ?) in Hoare's CSP [15]. If process A executes

$$x \leftarrow exchange(y)$$

then it waits until process B is also executing an *exchange* operation. If B never does so, then A is stuck—we must prove that this cannot happen. When B is at

$$u \leftarrow exchange(v)$$

then the two operations execute simultaneously, with x getting the value of v and u getting the value of y. More details of the *exchange* operation may be found in the next section and in Section VII.

V. History Variables and Safety Properties

In this section we present some notation for dealing with history variables. We use this notation to specify the safety properties of the *read*, *write*, and *exchange* operations.

Suppose X and Y are history variables. We write

$$X \leqslant Y$$

to denote that X is a *prefix* (initial subsequence) of Y. This means that $|X| \leqslant |Y|$, and the two sequences are identical in their first $|X|$ elements, where $|X|$ denotes the length of history X.

The initial value of a history variable is the empty sequence, and the only operation allowed is appending a new value. If history X has elements a, b, c, d then we can write

$$X = \langle abcd \rangle.$$

If $|X| = n$ then we may also write

$$X = \langle x_i \rangle_{i=1}^{n}.$$

We denote concatenation of sequences by juxtaposition, that is,

$$X = \langle ab \rangle \langle cd \rangle = \langle abcd \rangle.$$

We will speak about sequences of repeated messages. We use the superscripts * and $^+$ as defined for regular expressions:

$$X \in \langle x^* \rangle \equiv \exists k \geqslant 0 (X = \langle x^k \rangle)$$

$$X \in \langle x^+ \rangle \equiv \exists k \geqslant 1 (X = \langle x^k \rangle)$$

where $\langle x^k \rangle$ denotes the message x repeated k times. These notations can also be combined. For example

$$X \in \langle x_i^+ \rangle_{i=1}^{n}$$

means that X is a sequence that starts with one or more instances of x_1, followed by one or more instances of x_2, and so on, ending with one or more instances of x_n.

We use history variables to specify the safety properties of the *read*, *write*, and *exchange* operations. These specifications are in the form

$$\{P\}\ S\ \{Q\}.$$

This formula states that if P is true before statement S is executed, and statement S terminates, then Q is true on termination. The predicate P is called the precondition; predicate Q is called the postcondition. The safety specification for the *read* operation is

$$\{\sigma = \langle s_i \rangle_{i=1}^{n}\}\ x \leftarrow S.read\ \{\exists a (\sigma = \langle s_i \rangle_{i=1}^{n} \langle a \rangle \wedge x = a)\},$$

which states that if the input history contains n elements and a read is performed, then the input history will contain $n + 1$ elements (the first n being the same as before), and x will contain the new element of the history. Similarly, the specification for the *write* operation is

$$\{\rho = \langle r_i \rangle_{i=1}^{n} \wedge\ x = a\}\ R.write(x)\ \{\rho = \langle r_i \rangle_{i=1}^{n} \langle a \rangle\}.$$

The specification of the *exchange* operation is a little more complicated. Let X be the history of bits sent to the medium by process P, and let Y be the history of bits received from the medium by P. Then, the specification for the *exchange* operation is

$$\{X = \langle x_i \rangle_{i=1}^{n} \wedge Y = \langle y_j \rangle_{j=1}^{k} \wedge\ w = a\}$$

$$u \leftarrow exchange(w)$$

$$\{X = \langle x_i \rangle_{i=1}^{n} \langle a \rangle \wedge \exists b (Y = \langle y_j \rangle_{j=1}^{k} \langle b \rangle \wedge u = b)\}.$$

These conditions do not fully specify the behavior of the medium under an *exchange* operation, however. We must also specify that the lengths of all four histories are equal (because the pair of exchanges operates on all four simultaneously) and that the output histories look like the input histories. We state these properties as invariants of the medium:

$$|\alpha| = |\beta| = |\gamma| = |\delta|$$

$$Lose(\alpha, \beta) \wedge Lose(\gamma, \delta)$$

where $Lose\ (X, Y)$ is true if

$$\forall i \leqslant |Y|((y_i = \lambda) \vee (y_i = x_i)).$$

The function *Lose* compares corresponding positions of X and Y and is true if the bits in those positions are equal or if the bit in Y represents a lost message. These invariants define a medium that corresponds to the informal definition given in Section II.

VI. Proof of a Simplified System

To verify the AUY protocol using the abstract-program approach, we must first translate the finite-state machines into abstract programs. Fig. 5 shows a naive translation. It is naive in the sense that the automaton has been translated exactly: choices of transitions become if-then-else statements, transitions to the same node become while-do statements, and no transition for a given state becomes an error statement. Our eventual goal is to prove that the output history is a prefix of the input history:

$$\rho \leqslant \sigma.$$

We will discuss only safety properties in detail in this section. Section VIII will discuss liveness properties.

We called the translation naive, because there are some obvious simplifications that we can make to the code. We will transform this naive code by proving invariant properties of the processes that will allow us to eliminate unnecessary tests. Although this transformation process is informal, it could be stated formally. Scherlis [26] has formalized similar transformations for Lisp.

We will note that process B only transmits 1's and λ's (never 0's). This invariant is stated as

$$\forall g \in \gamma((g = 1) \vee (g = \lambda)).$$

With this invariant we can simplify the transmitter by eliminating one error check and changing the first check of state q to checking for an acknowledgment of 1 rather than anything but λ. Fig. 6 shows the revised code annotated with assertions. Note that we have also used the fact that the first elements of α and γ are both equal to λ, and hence, the first messages received are λ's as well. We have also assumed that none of the input from S equals λ.

We observe next that if process B receives a λ in one exchange, then it sends a λ in the next exchange; furthermore, if it receives a 0 or a 1, then it sends a 1 on the next exchange. We state this property with the following two invariants:

$$(|\beta| \geqslant k \geqslant 2 \wedge \beta_{k-1} = \lambda) \supset \gamma_k = \lambda$$

$$(|\beta| \geqslant k \geqslant 2 \wedge \beta_{k-1} \neq \lambda) \supset \gamma_k = 1$$

where γ_k is the kth element of γ. Because the medium always passes λ unchanged, these invariants allow us to simplify the transmitter code. These invariants describe a fundamental insight into the operation of the protocol: the receiver acts as an echoing device. If the transmitter receives a nonempty acknowledgment, it knows that the message it sent during the last exchange was in fact received. In Fig. 7 we apply

```
Process A:
 Init: ack ← exchange(λ)
       inputval ← S.read
 Loop
  P: While ack = 1 Do
       ack ← exchange(λ)
       inputval ← S.read
     Od
     If ack = 0 Then error Fi
     ack ← exchange(inputval)
  Q: If ack ≠ λ Then error Fi
     ack ← exchange(λ)
 Endloop
Endprocess

Process B:
 Init: msg ← exchange(λ)
 Loop
  R: If msg = λ Then
       msg ← exchange(λ)
     Else
       msg ← exchange(1)
     Fi
  S: While msg ≠ λ Do
       R.write(msg)
       msg ← exchange(1)
     Od
     msg ← exchange(λ)
 Endloop
Endprocess
```

Fig. 5. Naive translation.

```
Process A:
 Init: ack ← exchange(λ)
       inputval ← S.read
       { ack = λ & last(α) = λ }
 Loop
       { last(α) = λ }
  P: While ack = 1 Do
       { ack = 1 & last(α) = λ }
       ack ← exchange(λ)
       inputval ← S.read
       { last(α) = λ }
     Od
     { ack = λ & last(α) = λ }
     ack ← exchange(inputval)
     { last(α) ≠ λ }
  Q: If ack = 1 Then error Fi
     { last(α) ≠ λ & ack = λ }
     ack ← exchange(λ)
     { last(α) = λ }
 Endloop
Endprocess

Process B:
 Init: msg ← exchange(λ)
       { msg = λ & last(γ) = λ }
 Loop
       { last(γ) = λ }
  R: If msg = λ Then
       { last(γ) = λ & msg = λ }
       msg ← exchange(λ)
       { last(γ) = λ }
     Else
       { last(γ) = λ & msg ≠ λ }
       msg ← exchange(1)
       { last(γ) = 1 }
     Fi
  S: While msg ≠ λ Do
       { msg ≠ λ }
       R.write(msg)
       msg ← exchange(1)
       { last(γ) = 1 }
     Od
     { msg = λ }
     msg ← exchange(λ)
     { last(γ) = λ }
 Endloop
Endprocess
```

Fig. 6. Annotated translation. After proving that B never sends 0 as an acknowledgment, we can modify A's program. The notation "last(x)" indicates the last element of history x.

```
Process A:
 Init: ack ← exchange(λ)
       inputval ← S.read
       { ack = λ & last(α) = λ }
 P: While ack = 1 Do
     { cannot happen because ack = λ }
    Od
    { ack = λ & last(α) = λ }
    ack ← exchange(inputval)
    { last(α) ≠ λ  & ack = λ }
 Q: If ack = 1 Then { cannot happen } Fi
    { last(α) ≠ λ & ack = λ }
    ack ← exchange(λ)
    { last(α) = λ }
 Loop
     { last(α) = λ }
  P: While ack = 1 Do
      { ack = 1 & last(α) = λ }
      ack ← exchange(λ)
      inputval ← S.read
      { last(α) = λ & ack = λ }
      { therefore the while can only be executed once }
     Od
     { ack = λ & last(α) = λ }
     ack ← exchange(inputval)
     { last(α) ≠ λ & ack = λ }
  Q: If ack = 1 Then { cannot happen } Fi
     { last(α) ≠ λ & ack = λ }
     ack ← exchange(λ)
     { last(α) = λ }
 Endloop
Endprocess
```

Fig. 7. Annotated transmitter. After we know that B "repeats" what A sent the previous exchange, we can determine when ack = λ.

```
Process A:
 Init: ack ← exchange(λ)
       inputval ← S.read
       { ack = λ & last(α) = λ }
 P: ack ← exchange(inputval)
    { last(α) ≠ λ  & ack = λ }
 Q: ack ← exchange(λ)
    { last(α) = λ }
 Loop
     { last(α) = λ }
  P: If ack = 1 Then
      { ack = 1 & last(α) = λ }
      ack ← exchange(λ)
      inputval ← S.read
      { last(α) = λ & ack = λ }
     Fi
     { ack = λ & last(α) = λ }
     ack ← exchange(inputval)
     { last(α) ≠ λ & ack = λ }
  Q: ack ← exchange(λ)
     { last(α) = λ }
 Endloop
Endprocess
```

Fig. 8. Annotated transmitter. Using the information from the previous figure allows us to eliminate some more code. Note that all error conditions are gone.

these invariants. Note that we have unrolled the first iteration of the main loop. Fig. 8 shows the result of simplifying the code after applying the invariants.

For the remainder of this section, we will assume that no messages are lost. Analyzing this simpler problem will help us understand the more general problem. Our new assumption may be written as

$$(\alpha = \beta) \wedge (\gamma = \delta).$$

Using this assumption we may again simplify process A: if process B sends a 1 we know that it will get through to A. Fig. 9 shows the result of this simplification. Note that there are no more tests (if statements or while statements) left. Therefore, we can describe the history α (and hence β) independently of the receiver:

```
Process A:
 Init: ack ← exchange(λ)
       inputval ← S.read
       { ack = λ & last(α) = λ }
 P: ack ← exchange(inputval)
    { last(α) ≠ λ  & ack = λ }
 Q: ack ← exchange(λ)
    { last(α) = λ & ack = 1 }
 Loop
     { last(α) = λ & ack = 1 }
  P: ack ← exchange(λ)
     inputval ← S.read
     { last(α) = λ & ack = λ }
     ack ← exchange(inputval)
     { last(α) ≠ λ & ack = λ }
  Q: ack ← exchange(λ)
     { last(α) = λ & ack = 1 }
 Endloop
Endprocess
```

Fig. 9. Annotated transmitter (no messages lost). If no messages are lost then the program above is the result. Note that there are no tests left.

$$\sigma = \langle s_i \rangle_{i=1}^{n} \supset (\langle \lambda s_i \lambda \rangle_{i=1}^{n-1} \langle \lambda \rangle \leqslant \alpha \leqslant \langle \lambda s_i \lambda \rangle_{i=1}^{n} \langle \lambda \rangle).$$

Now that we know the value of β, we can derive the value of ρ from the receiver code:

$$(\langle \lambda s_i \lambda \rangle_{i=1}^{n-1} \langle \lambda s_n \rangle \leqslant \beta \leqslant \langle \lambda s_i \lambda \rangle_{i=1}^{n} \langle \lambda \rangle) \supset \langle s_i \rangle_{i=1}^{n-1} \leqslant \rho \leqslant \langle s_i \rangle_{i=1}^{n}.$$

This then proves that the output is a prefix of the input.

We give no formal liveness proof in this section. Informally, though, the only place that a process gets blocked is at an *exchange* when the other process never gets to an *exchange*. Nothing can stop either process from repeatedly reaching an *exchange*. Because one message is sent each loop of B, the system is live—the length of the output will always keep increasing. This reasoning can be stated formally using the notation of temporal logic.

Fig. 10 shows a final condensed version of the code for process A.

VII. Temporal Logic and Liveness Properties

In this section we present a brief introduction to temporal logic. We use temporal logic to specify the liveness properties of the *read*, *write*, and *exchange* operation. For more details on the theory of temporal logic and the use of temporal logic in program verification, see the work of Pnueli, Lamport, Owicki, and Hailpern [11], [18], [23]–[25]. Recently, many papers have used temporal logic to verify network protocols [6], [19], [28], [29].

Temporal logic provides operators for reasoning about the future, where the future is a program computation (a sequence of states that could arise during program execution). Informally, the first state in a computation represents the present; subsequent states represent the future. The temporal logic we are using has two operations, □ (henceforth) and ◇ (eventually). The formula □P (henceforth P) means "P is true for all states in the computation." The formula ◇P is interpreted as "there is some state in the computation in which P is true." When we say that a temporal formula is true for a program, we mean that it is true for all computations of that program. Note that the present is considered to be part of the future in this temporal logic.

Temporal operators are particularly useful for stating liveness properties. For example, program termination can be expressed by

$$at\ P \supset \Diamond after\ P$$

where *at* P and *after* P are assertions that are true of states

```
Process A:
 Loop
   ack ← exchange(λ)
   inputval ← S.read
   ack ← exchange(inputval)
   ack ← exchange(λ)
 Endloop
Endprocess
```

Fig. 10. Condensed transmitter (no messages lost).

in which control is at the beginning or end of program P, respectively.

Combinations of the two temporal operators are also useful. The formula $\Box\Diamond P$ ("infinitely often P") implies that there are an infinite number of future states for which P is true. This combination is useful for stating recurring properties of a program, for example,

$\Box\Diamond$(the buffer is not full).

The formula $\Diamond\Box P$ states that there is some point in the future at which P becomes true and remains true thereafter. An example of this combination is to state that deadlock is inevitable:

$\Diamond\Box$(all processes are waiting).

There are two temporal assertions about histories that we will use in reasoning about liveness. The first is an assertion that the size of a given history will grow without bound. It is abbreviated as $u(X)$, where

$$u(X) = \forall n(\Diamond(|X| > n)).$$

The second assertion states that a particular value occurs an unbounded number of times in the history. Letting $c(X, b)$ be the number of occurrences of bit b in history X, we have

$$uc(X, b) = \forall n(\Diamond(c(X, b) > n)).$$

We use this notation to describe how *error-prone* the medium is. If all messages and acknowledgments were lost, then the system could not be live. The definition of robust required that six exchanges in a row not be lost. We make a weaker assumption: if b is sent an unbounded number of times, then b is received correctly an unbounded number of times. This is expressed as

$$uc(\alpha, b) \supset uc(\beta, b)$$

$$uc(\gamma, b) \supset uc(\delta, b).$$

Our basic liveness assumption concerning the execution of program statements is that they do not block:

$$at\ s \supset \Diamond after\ s$$

where s is any statement that does not affect shared variables. We have assumed that the *read* and *write* statements follow this rule as well; that is, if a process is attempting to read from or write to an outside user, then that operation will eventually terminate. This last statement is an assumption on the behavior of the sending and receiving users.

The situation is more complex when we specify the liveness properties of the *exchange* operation, because both processes are involved. Our first liveness statement about the *exchange* operation is

$$(A\ at\ L1{:}exchange \wedge B\ at\ L2{:}exchange) \supset \Diamond(A\ after\ L1 \wedge B\ after\ L2),$$

that is, if both A and B are ready to execute an *exchange* ($L1$ and $L2$ are labels to distinguish the *exchange* operations in a process), then both processes will eventually finish their corresponding operations. We must also preclude one process finishing an *exchange* without the other:

$$(A\ at\ L{:}exchange \wedge \neg\Diamond B\ at\ exchange) \supset \neg\Diamond A\ after\ L$$

and similarly for B.

VIII. Proof (Outline) of the AUY Protocol

We now return to the proof of the AUY protocol. Fig. 11 shows the protocol before we made our simplifying assumption in Section VI. Process B is still rather complicated; proving properties of process A will allow us to simplify it somewhat. Note that process A never sends two non-λ messages in a row. This invariant may be stated as

$$(n \geqslant 2 \wedge \alpha = \langle a_i \rangle_{i=1}^{n}) \supset \neg\exists k \leqslant (n-1)(a_k \neq \lambda \wedge a_{k+1} \neq \lambda).$$

This invariant allows us to simplify the code for process B in two ways. First, the while loop can become an if statement, because if $msg \neq \lambda$ the first time through, then it must equal λ the second time (by the last invariant). Second, if the first if statement of process B determines that $msg \neq \lambda$, then the body of the second if statement (that used to be a while loop) will not be executed (again because of the invariant above). Fig. 12 shows the revised receiver code.

Let us turn again to process A. We can also relate the output A, α, to its input, (σ):

$$\sigma = \langle s_i \rangle_{i=1}^{n} \supset \alpha \leqslant \langle\langle \lambda s_i \lambda \rangle \langle s_i \lambda \rangle^* \rangle_{i=1}^{n}.$$

We can also show that A only starts sending a new bit when it receives an acknowledgment of 1 from B (note that the only place that a 1 can be received in the code of A is at the top of the loop). The key theorem we must prove is that the 1 acknowledgment guarantees that B has received the current bit and has output that bit to the receiving user.

What does an acknowledgment of 1 from B mean? That leads to the following question: how does B differentiate two distinct messages from A? If at the beginning of its loop, B sees a non-λ message, then it treats the message as a duplicate and sends a duplicate acknowledgment. If, however, the message at the top of the loop equals λ and the next message does not, then this second message is treated as new (output to the receiving user and acknowledged). In other words, between duplicate messages B expects an odd number of λ's; between different messages there are an even number of λ's. Hence, the parity (even/odd) of the length of β when a bit is received determines whether the bit is new or old. Furthermore, the number of changes in message parity encodes a sequence number for each message. To prove that acknowledgments received by A reflect that the correct message has been received by B, we must show that the two processes always agree on this encoded sequence number for all non-λ messages.

We define $\Delta(\alpha, i)$ to be the number of changes in parity in α up to and including element i; thus, it is the sequence number of the ith element in the sequence. We define $\Delta(\alpha, i)$ as follows. In the base case

$$\Delta(\alpha, 0) = 0.$$

Let α_i denote the ith element of alpha. If $\alpha_i = \lambda$ then

$$\Delta(\alpha, i) = \Delta(\alpha, i-1).$$

If i is even and $\alpha_i \neq \lambda$ then

$$\Delta(\alpha, i) = \begin{cases} \Delta(\alpha, i-1) + 1, & \text{if even } (\Delta(\alpha, i-1)) \\ \Delta(\alpha, i-1), & \text{if odd } (\Delta(\alpha, i-1)). \end{cases}$$

```
Process A:
 Init: ack ← exchange(λ)
       inputval ← S.read
       { ack = λ & last(α) = λ }
 P:  ack ← exchange(inputval)
     { last(α) ≠ λ  & ack = λ }
 Q:  ack ← exchange(λ)
     { last(α) = λ }
 Loop
      { last(α) = λ }
  P:  If ack = 1 Then
       { ack = 1 & last(α) = λ }
       ack ← exchange(λ)
       inputval ← S.read
       { last(α) = λ & ack = λ }
      Fi
      { ack = λ & last(α) = λ }
      ack ← exchange(inputval)
      { last(α) ≠ λ & ack = λ }
  Q:  ack ← exchange(λ)
      { last(α) = λ }
 Endloop
Endprocess

Process B:
 Init: msg ← exchange(λ)
       { msg = λ & last(γ) = λ }
 Loop
      { last(γ) = λ }
  R:  If msg = λ Then
       { last(γ) = λ & msg = λ }
       msg ← exchange(λ)
       { last(γ) = λ }
      Else
       { last(γ) = λ & msg ≠ λ }
       msg ← exchange(1)
       { last(γ) = 1 }
      Fi
  S:  While msg ≠ λ Do
       { msg ≠ λ }
       R.write(msg)
       msg ← exchange(1)
       { last(γ) = 1 }
      Od
      { msg = λ }
      msg ← exchange(λ)
      { last(γ) = λ }
 Endloop
Endprocess
```

Fig. 11. Annotated protocol.

```
Process B:
 Init: msg ← exchange(λ)
       { msg = λ & last(γ) = λ }
 Loop
  { last(γ) = λ }
  If msg = λ Then
   { last(γ) = λ & msg = λ }
   msg ← exchange(λ)
   { last(γ) = λ }
   If msg ≠ λ Then
    { msg ≠ λ & last(γ) = λ }
    R.write(msg)
    msg ← exchange(1)
    { last(γ) = 1 & msg = λ }
   Fi
   { msg = λ }
  Else
   { last(γ) = λ & msg ≠ λ }
   msg ← exchange(1)
   { last(γ) = 1 & msg = λ }
  Fi
  { msg = λ }
  msg ← exchange(λ)
  { last(γ) = λ }
 Endloop
Endprocess
```

Fig. 12. Annotated receiver. After proving that the transmitter cannot send two non-λ messages in a row, we can change the while loop into an if statement and move the result into the then-clause of the previous if statement.

Similarly, if i is odd and $\alpha_i \neq \lambda$ then

$$\Delta(\alpha, i) = \begin{cases} \Delta(\alpha, i-1) + 1, & \text{if odd } (\Delta(\alpha, i-1)) \\ \Delta(\alpha, i-1), & \text{if even } (\Delta(\alpha, i-1)). \end{cases}$$

We will denote the last sequence number in the history by $\Delta(\alpha)$, where

$$\Delta(\alpha) = \Delta(\alpha, |\alpha|).$$

We can define $\Delta(\beta)$ in the same way as $\Delta(\alpha)$. For γ and δ, Δ is defined in the same manner as above, but with the role of the even and odd values of i reversed (because acknowledgments come one time unit after receipt of a message).

This encoding of sequence number as changes in parity is not a new idea. We have proved the correctness of another protocol that uses the same mechanism: the alternating bit protocol [11], [13]. In the AUY protocol the parity resides in the number of messages that have been received; in the alternating bit protocol the parity is explicitly included in the message as a one-bit sequence number. Changes in parity occur when the sequence bits in successive messages change from 0 to 1 (or 1 to 0). The safety proofs, however, are the same for the two protocols when the correct definitions of parity are made. We therefore refer the reader to the proof of the alternating bit protocol as a model for the remainder of the safety proof of the AUY protocol.

The liveness proof closely follows the liveness proof of the alternating bit protocol. As in Section VI, neither process can block at an *exchange* operation, because both processes are infinitely often at exchanges. Our goal is to prove that the length of ρ will always increase:

$$u(\rho).$$

We present one liveness assertion for each of the two processes. These assertions can be proved directly from the code of the processes. Process A promises to start sending the next bit when the current one has been acknowledged:

$$\forall j(\Delta(\delta) \geq j \supset (uc(\alpha, \sigma_{j+1}) \vee \Diamond(\Delta(\delta) \geq j+1))).$$

Therefore, if process A receives acknowledgment j, then A starts to send bit $j+1$. It will send that bit an unbounded number of times unless it eventually receives acknowledgment $j+1$. The liveness assertion for process B is similar:

$$\forall j(\Delta(\beta) \geq j \supset (\Diamond(|\rho| \geq j) \wedge (uc(\gamma, 1) \vee \Diamond(\Delta(\beta) \geq j+1)))).$$

We prove $u(\rho)$ by induction on the length of ρ. The base step is to show $\Diamond(|\rho| \geq 0)$, which is true initially. The induction step involves showing that if ρ contains k messages at some point, then it will eventually contain $k+1$ messages:

$$\Box(|\rho| = k \supset \Diamond|\rho| > k).$$

The rest of this proof exactly parallels the liveness proof of the alternating bit protocol. To prove the induction step, we assume that $|\rho| = k$ and follow the alternatives as specified in the two liveness assertions above (along with our liveness assumption about the medium). The details of the proof may be found in [13].

IX. Conclusion

In proving the correctness of this protocol we notice that the finite-state approach was simple and the abstract-program approach was complex—why? One reason that the first approach is simple is that the pure finite-state technique does not address the issue of correct data transfer. Another reason is that the

finite-state description encodes much information in the set of current states. That information gets translated into control flow in the abstract-program representation. The significant pieces of information must then be extracted by the proof process.

Our development of the abstract-program proof was useful in its own right. We showed why the potential error conditions were not errors (because the receiver acted as an echoing device). We proved that the while loops in the finite-state machine could only be executed a single time. We discovered the underlying mechanism for the operation of the protocol (the even/odd parity of time that the bits were sent encoded a sequence number).

Acknowledgment

I would like to thank A. Johassen and D. Knuth [17] for the idea for the title of this paper, A. Aho for discussing his protocol with me, and S. Owicki for her insights into the operation of the protocol. Special thanks go to the referees for their comments and suggestions.

References

[1] A. V. Aho, J. D. Ullman, and M. Yannakakis, "Modeling communications protocols by automata," in *Proc. 20th Annu. IEEE Symp. Foundations Comput. Sci.*, San Juan, PR, Oct. 1979, pp. 267–273.

[2] G. V. Bochmann, "Communication protocols and error recovery procedures," *ACM Oper. Syst. Rev.*, vol. 9, pp. 45–50, July 1975.

[3] G. V. Bochmann and C. A. Sunshine, "Formal methods in communication protocol design," *IEEE Trans. Commun.*, vol. COM-28, pp. 624–631, Apr. 1980.

[4] G. V. Bochmann, "A general transition model for protocols and communication services," *IEEE Trans. Commun.*, vol. COM-28, pp. 643–650, Apr. 1980.

[5] ——, "A hybrid model and the representation of communication services," in *Computer Network Architectures and Protocols*, P. E. Green, Ed. New York: Plenum, 1982, pp. 625–644.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach," in *Proc. 10th Annu. ACM Symp. Principles of Programming Languages*, Austin, TX, Jan. 1983, pp. 117–126.

[7] A. A. S. Danthine, "Protocol representation with finite state models," in *Computer Network Architectures and Protocols*, P. E. Green, Ed. New York: Plenum, 1982, pp. 579–606.

[8] R. W. Floyd, "Assigning meanings to programs," in *Proc. Symp. Appl. Math.*, vol. XIX, Amer. Math. Soc., 1967, pp. 19–32.

[9] D. I. Good and R. M. Cohen, "Principles of proving concurrent programs in Gypsy," in *Proc. 6th Annu. ACM Symp. Principles of Programming Languages*, San Antonio, TX, Jan. 1979, pp. 42–52.

[10] M. G. Gouda, "On "A simple protocol whose proof isn't": The state machine approach," this issue, pp. 382–384; also Dep. Comput. Sci., Univ. Texas at Austin, Tech. Rep. TR-84-21, July 1984.

[11] B. T. Hailpern, *Verifying Concurrent Processes Using Temporal Logic* (Lecture Notes in Computer Science, vol. 129). New York: Springer-Verlag, 1982; also, Ph.D. dissertation, Stanford Univ., Stanford, CA, 1980.

[12] ——, "Specifying and verifying protocols represented as abstract programs," in *Computer Network Architectures and Protocols*, P. E. Green, Ed. New York: Plenum, 1982, pp. 607–624.

[13] B. T. Hailpern and S. Owicki, "Modular verification of computer communication protocols," *IEEE Trans. Commun.*, vol. COM-31, pp. 56–68, Jan. 1983.

[14] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, pp. 576 *ff.*, May 1969.

[15] ——, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666–677, Aug. 1978.

[16] J. H. Howard, "Proving monitors," *Commun. ACM*, vol. 19, pp. 273–279, May 1976.

[17] A. T. Johassen and D. E. Knuth, "A trivial algorithm whose analysis isn't," *J. Comput. Syst. Sci.*, vol. 16, pp. 301–322, June 1978.

[18] L. Lamport, " 'Sometimes' is sometimes 'not never': On the temporal logic of programs," in *Proc. 7th Annu. ACM Symp. Principles of Programming Languages*, Las Vegas, NV, Jan. 1980, pp. 174–185.

[19] ——, "Specifying concurrent program modules," *ACM Trans. Programming Languages Syst.*, vol. 5, pp. 190–222, Apr. 1983.

[20] S. S. Owicki and D. Gries, "Verifying properties of parallel programs: An axiomatic approach," *Commun. ACM*, vol. 19, pp. 279–285, May 1976.

[21] S. S. Owicki, "Specifications and proofs for abstract data types in concurrent programs," in *Program Construction*, F. L. Bauer and M. Broy, Eds. New York: Springer-Verlag, 1979, pp. 174–197.

[22] ——, "Specifications and verification of a network mail system," in *Program Construction*, F. L. Bauer and M. Broy, Eds. New York: Springer-Verlag, 1979, pp. 198–234.

[23] S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," *ACM Trans. Programming Languages Syst.*, vol. 4, pp. 455–495, July 1982.

[24] A. Pnueli, "The temporal logic of programs," in *Proc. 18th Annu. IEEE Symp. Foundations Comput. Sci.*, Providence, RI, Oct. 1977, pp. 46–57.

[25] ——, "The temporal semantics of concurrent programs," in *Semantics of Concurrent Computation*. New York: Springer-Verlag, 1979, pp. 1–20.

[26] W. L. Scherlis, "Expression procedures and program derivation," Ph.D. dissertation, Stanford Univ., Stanford, CA, Comput. Sci. Tech. Rep. STAN-CS-80-818, 1980.

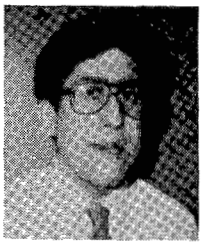
[27] R. Schwartz and P. M. Melliar-Smith, "From state machines to temporal logic: Specification methods for protocol standards," *IEEE Trans. Commun.*, vol. COM-30, pp. 2486–2497, Dec. 1982.

[28] R. L. Schwartz, P. M. Melliar-Smith and F. H. Vogt, "An interval logic for higher-level temporal reasoning," *in Proc. 2nd Annu. ACM Symp. Principles of Distributed Comput.*, Montreal, P.Q., Canada, Aug. 1983, pp. 173–186.

[29] D. E. Shasha, A. Pnueli, and W. Ewald, "Temporal verification of carrier-sense local area network protocols," IBM, Yorktown Heights, NY, Res. Rep. RC 10132, Aug. 1983.

[30] C. A. Sunshine, "Formal techniques for protocol specification and verification," *IEEE Computer*, vol. 12, pp. 20–27, Sept. 1979.

★



Brent Hailpern (S'80–M'80–SM'84) was born in Denver, CO, on January 11, 1955. He received the B.S. degree in mathematics from the University of Denver, Denver, CO, 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 workstation languages and protocols. His research interests include concurrent programming, program verification, network protocols, and distributed systems.

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