COMPUTER SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES
DEPARTMENT OF ELECTRICAL ENGINEERING
STANFORD UNIVERSITY · STANFORD, CA 94305

VERIFYING NETWORK PROTOCOLS USING TEMPORAL LOGIC

Brent T. Hailpern

Susan S. Owicki

TECHNICAL REPORT NO. 192

June 1980

This work was supported by the Joint Services Electronics Project under contract N-00014-75-C-0601. Brent Hailpern was supported by fellowships from the National Science Foundation and the Fannie and John Hertz Foundation

VERIFYING NETWORK PROTOCOLS USING TEMPORAL LOGIC

Brent T. Hailpern
Susan S. Owicki

TECHNICAL REPORT NO. 192

June 1980

COMPUTER SYSTEMS LABORATORY
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

ABSTRACT

Programs that implement computer communications protocols can exhibit extremely complicated behavior, and neither informal reasoning nor testing is reliable enough to establish their correctness. In this paper we discuss the application of program verification techniques to protocols. This approach is more reliable than informal reasoning, but has the advantage over formal reasoning based on finite-state models that the complexity of the proof does not grow unmanageably as the size of the program increases. Certain tools of concurrent program verification that are especially useful for protocols are presented: history variables that record sequences of input and output values, temporal logic for expressing properties that must hold in a future system state (such as eventual receipt of a message), and module specification and composition rules. The use of these techniques is illustrated by verifying a simple data transfer protocol from the literature.

KEYWORDS: Verification, Concurrency, Proof of correctness, Specifications, Networks, Network Protocols, Temporal Logic.

Introduction

Programs that implement computer communication protocols can exhibit extremely complicated behavior, since 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 may be found in Sunshine [17]. Most previous approaches to verifying network protocols have been based upon reachability arguments for finite-state models of the protocols. However, only protocols of limited complexity can be verified using finite-state models, because of the combinatorial explosion of the state space as the complexity of the protocol increases. Finite-state models also have difficulty in expressing properties related to correct data transfer. 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 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.

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 [2, 7]. 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 may 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. Since 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 non-terminating programs like operating systems. Aside from sequential termination, there has been little work on the verification of liveness properties. Since 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 liveness, we use the notation of temporal logic [13], which provides operators for making assertions about future program states.

In this paper we present the verification of a simple data transfer protocol to illustrate the application of concurrent program verification techniques. We first describe the protocol to be verified, then discuss two of our basic tools: temporal logic and history sequences. The rest of the paper is devoted to a modular proof of the safety and liveness properties of the protocol.

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 [16]. Stenning verified the safety properties of the algorithm, using a non-modular proof technique. He did not consider liveness, and, in fact, it is possible for the algorithm to enter an infinite loop and fail to deliver messages. In the version presented here, that fault *has* been repaired, and we give a proof of its liveness.

Figure 1 contains the code for the simplified Stenning protocol to be considered, and Figure 2 is a diagram illustrating the network structure. The verification of the full protocol is given in Hailpern [6]. The code consists of two processes: a transmitter and a receiver. The transmitter takes as input an unbounded sequence of messages, X. It sends them to the receiver, via the communication medium mtr. The receiver outputs messages to the output sequence, Y, and acknowledges receipt via the communication medium mrt. Complications arise because the communication media are unreliable. Messages can be lost, duplicated, or reordered. (We assume 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.

Temporal Logic

Temporal logic provides operators for reasoning about program computations. A computation 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 [4, 13, 14, 15], and is further described by Lamport [9]. Its two basic operators are $\Box$ (henceforth) and *o* (eventually). The formula $\Box P$ (henceforth P) means "P is true for all states in the computation." The formula $\Diamond P$ (eventually P) is interpreted as "there is some state in the computation in which P is true." The modalities $\Box$ and *o* are duals, that is,

$$\Box P \equiv \sim \Diamond \sim P.$$

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

$$at\ P \quad 3 \quad \Diamond\ after\ P,$$

where *at 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

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

Combinations of the two modalities are also useful. For example, the formula $\Box\Diamond$ *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 $\Box\Diamond P$ 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 $\Box\Diamond$ operator is especially useful for stating recurring properties of a program, for example,

$$\Box\Diamond(\textit{the buj jet is not } \textit{full})$$

The dual of infinitely often ($\Box\Diamond$) is $\Diamond\Box$, pronounced eventually *henceforth*. The formula $\Diamond\Box P$ states that there is some point after which P remains true forever. An example of the use of this modality concerns program deadlock. Deadlock occurs when each process in a system is waiting for some other process to release a resource. One could state that deadlock is inevitable by the formula

$$\Diamond\Box(\textit{all pr ocesses are waiting}).$$

Histories

Our proof uses 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 [5, 8, 11, 12]. The -names of the history variables are indicated in the network diagram in Figure 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 a and β, respectively, while the input and output histories of the acknowledgment medium are γ and 6. We denote the i^{th} element of X by d_i, since it is the i th datum. A message consisting of the pair i and d_i, that is $[i, d_i]$, will be abbreviated M_i. This is the form of messages in α and β. An acknowledgment for message i, the pair $[i, \text{"ack"}]$ is denoted by A_i: messages in γ and δ have this form.

We now introduce some notation for describing histories. Let A and B be arbitrary history sequences. The length of A is denoted by $|A|$. If A has elements u, v, y, z then we can write,

$$A = < uvyz > .$$

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

$$A = < a_i >_{i=1}^{n} .$$

We denote concatenation of sequences by juxtaposition, that is,

$$A = < uv >< yz > = < uvyz >$$

We will often speak about sequences of repeated messages in referring to the histories of messages sent and received. We make use of the superscripts * and + as defined for regular expressions:

$$A = < a^+ > \equiv (\exists k \geq 0)(A = < a^k >)$$
$$A = < a^+ > \equiv (\exists k \geq 1)(A = < a^k >)$$

where $< a^k >$ denotes the message $< a >$ repeated k times. These notations can be combined, for example

$$A = < a_i^+ >_{i=1}^n$$
$$= < a_1^+ >< a_2^+ > \cdots < a_n^+ >.$$

We write $A \preceq B$, if A is an *initial subsequence* of B. This means that $|A| \leq |B|$, and the two sequences are identical in their first $|A|$ elements.

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

$$u(A) = \forall n(\Diamond |A| > n),$$

or equivalently

$$u(A) = \forall n(\Box(|A| = n \Rightarrow \Diamond |A| > n)).$$

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

$$uc(A, m) = \forall n(\Diamond c(A, m) > n).$$

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, and these might 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.

The medium specification involves three kinds of information. First, there is an *invariant*, an assertion about the medium that is true at all times in the computation. The invariant describes the safety properties of the medium. Second,

liveness properties are specified by temporal logic assertions called commitments. Commitments describe conditions that the process causes to become true. Finally, the services provided by the medium (sending and receiving a message, checking for the presence of messages) must be precisely defined.

Since the communication medium we are considering is an unreliable one, it has a very weak invariant: nothing comes out that was not put in. More precisely,

$$m \in \beta \quad \supset \quad m \in \alpha \qquad (mtr1)$$

$$m \in \delta \quad \supset \quad m \in \gamma \qquad (mrt1)$$

where m is any message. This safety assertion describes a medium which may lose, duplicate, and re-order 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 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.

$$u(\alpha) \quad \supset \quad \Box\Diamond\, mtr.ExistsM \qquad (mtr2)$$

$$u(\gamma) \supset 0 \quad 0 \quad mrt.ExistsM \qquad (mrt2)$$

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 is expressed by the formulas

$$(uc(\alpha, m)\, A\, u(\beta)) \quad \supset \quad Om \in \beta \qquad (mtr3)$$

$$(uc(\gamma, m)\, A\, u(6)) \quad \supset \quad \Diamond m \in \delta \qquad (mrt3).$$

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

To precisely specify the services of the medium we give *pre*, *post* and *live* assertions about each operation. Pre and post give safety (partial correctness) conditions for the operations. If the pre-condition holds when the operation is invoked, and if it terminates, then at termination the post-condition must hold. The variables in these assertion must be private to the process that invokes the operation, in the sense that no other process can modify their values. This avoids the complexity that can arise when dealing with variables that are shared between

several processes. (A general discussion of private variables is given by Owicki [11].) The live assertion describes the effects that the operation causes when it is invoked. This may be a conditional commitment (as shown for receive below) and it may involve variables that can be modified by other processes.

The medium we are assuming provides three services: 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)

pre: $a = A$
post: $a = A < m >$
live: at *mtr.send* $\supset$ O*after mtr.send*

receive(var m)

pre: $\beta = B$
post: $\beta = B < m >$
live: $(at\ mtr.receive \wedge \Diamond mtr.ExistsM) \supset$
O(*after mtr.receive*)

existsM

pre: *true*
post: *true*
live: *at mtr.existsM* $\supset$ $\Diamond$*after mtr.existsM*

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

The timer is another black box, and we could define its properties in a similar way. However, since 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.

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.

$$\exists n(X = < d_i >_{i=1}^{n} \quad \wedge \quad \alpha = < M_i^{+} >_{i=1}^{n-1} < M_n^{*} >) \qquad (T1)$$

$$|X| \geq k \quad \supset \quad A_{k-1} \in \delta \qquad (T2)$$

The first invariant states that when n items have been input to the transmitter, the output to the mtr medium 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 just after the operation read(X) in the transmitter, the n^{th} piece of data has been read in, but the message M_n has not yet been sent out. To see that assertion T1 is invariant, note that it holds initially (when all sequences are empty) and that it is preserved by each -operation of the transmitter. A formal proof that the transmitter maintains these invariants would include reasoning about the transmitter's changes to α and δ, using the the pre and post conditions of mtr.send and mrt.receive given in the last section.

The invariant T2 states that the k^{th} input item 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 that are similar to those of the transmitter

$$\begin{aligned}&(m \in \beta \supset \exists j(m = M_j)) \quad \supset \\ &\quad (\exists n(Y \preceq < d_i >_{i=1}^{n}) \ \wedge \ d_k \in Y \ \supset \ M_k \in \beta),\end{aligned} \qquad (R1)$$

$$A_i \in \gamma \quad \supset \quad (M_i \in \beta \wedge |Y| \geq i) \qquad (R2)$$

Roughly speaking, the invariant R1 states that the receiver output Y will be legitimate if its input β is legitimate. More precisely, if β contains only messages of the form $[i, d_i]$, then Y will be a sequence of data items $< d_i >_{i=1}^{n}$, and each datum in Y must correspond to a message that appears in β. 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. Since each message in β has the form $[i, d_i]$, the i^{th} element of Y must be d_i.

The second invariant states that if acknowledgment i is in the output history γ, message i is in the input history β, 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.

Safety: System

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

$$Y \preceq X. \qquad (S1)$$

This assertion states that the output values are some 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.

The system safety specifications are not verified by examining the program code. Instead, we show that the invariant of the system is implied by the invariants of its components. 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.

Let us now 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**

$$m \in \beta \quad \supset \quad \exists i(m = M_i)$$

follows immediately from the safety properties of the transmitter and receiver. Since 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 (mtrl), the receiver can only obtain legitimate messages.

Now let

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

Since the hypothesis of **R1** is satisfied, we know that the conclusion of **R1** holds, namely

$$Y \preceq < d_i >_{i=1}^{n}.$$

But by mtrl,

$$M_n \in \beta \quad \supset \quad M_n \in \alpha$$

and by T1,

$$M_n \in a \quad \supset \quad (< d_i >_{i=1}^{n} \preceq X)$$

Thus we can conclude

$$Y \preceq < d_i >_{i=1}^{n} \preceq X$$

which implies the system safety assertion S1.

Liveness: **Transmitter and Receiver**

Liveness specifications of processes, like those of monitors, are given by commitments. Verifying that a process satisfies its liveness specifications requires reasoning based on assumptions about the liveness properties of program statements. 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 be the assertion that s is ready to be executed, and *after* s the assertion that control has finished s. Our basic liveness assumption can be expressed in temporal logic by

$$at\ s \quad \supset \quad \Diamond a jter \quad 8.$$

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 (of mtr) states that receive will terminate if a message is available, that is,

$$(at\ mtr.receive \wedge mtr.ExistsM) \quad \supset \quad \Diamond a\ jter\ mtr.receive$$

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

loop S 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

$$at\ S \supset \Box \quad \Diamond atS,$$

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

$$(at\ S' \wedge \Box\ Omtr.EzdstsM) \supset \Box\Diamond at\ S'.$$

The assertion $\Box$ *omtr.Ezists 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 post condition of send, plus the fact that send can never block, we can conclude

$$(at\ mtr.send \wedge |a| = k) \supset \Diamond(after\ mtr.send \wedge |a| = k + 1).$$

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

$$\Box(|a| = k \supset \Diamond|a| = k + 1),$$

which implies u(a). Formal rules for proving liveness properties from program code are given by Lamport and Owicki [10]. 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 [6].

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

$$u(a) \qquad (T3)$$

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 known to be available. Given that there is no blocking, the timeout mechanism guarantees that messages are sent out at least once every timer interval.

The second transmitter commitment is

$$\Box\Diamond(mrt.Exists\ M) \supset u(6) \qquad (T4)$$

This states that the transmitter will increase the size of δ provided that the environment keeps making acknowledgments available in mrt. 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.

$$\forall i(A_i \in \delta \supset |X| \geq i) \quad \supset \quad \forall j(A_j \in \delta \supset (uc(a, M_{j+1}) \vee \diamond(A_{j+1} \in \delta))) \tag{T5}$$

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 β and 7 to grow unboundedly as long as it is able to receive messages from mtr.

$$\square \quad omtr.Ezists\ M \quad \supset \quad u(\gamma) \tag{R3}$$

$$\square\diamond mtr.ExistsM \quad \supset \quad u(\beta) \tag{R4}$$

The receiver code satisfies these assertions because the repeated availability of messages implies that the receiver can not be blocked at its receive operation. Therefore it repeatedly executes its loop body, and each time it increases the length of β and γ. 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 α 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.

$$(\forall i(M_i \in \beta \supset |Y| \geq i - 1) \wedge u(\beta)) \quad \supset \quad \forall j[\, M_j \in \beta \quad \supset \quad (\diamond(|Y| \geq j) \wedge (uc(\gamma, A_j) \ \vee \diamond M_{j+1} \in \beta))] \tag{R5}$$

This is analogous to transmitter commitment T5. Assuming that message i does not arrive until the receiver has processed $i - 1$, and that β *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, since it can never block.)

Liveneas: no starvation

Our first step in establishing system liveness is a proof that all of the medium history variables grow unboundedly.

$u(\alpha)$	(T3)
$u(\alpha) \supset \Box$ *O(mtr.EzistsM)*	(mtr2)
$\Box$ *O(mtr.EzistsM) 3 u(7)*	(R3)
$\Box$ *o(mtr.Exists M)* $\supset u(\beta)$	(R4)
u(7) $\supset \Box\Diamond(mrt.ExistsM)$	(mrt2)
$\Box$ *o(mrt.ExistsM) 3 u(6)*	(R3)

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

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

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

Liveness: System

The system liveness property we ultimately want to prove is that each message is eventually output. Since 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,

$$u(Y) \qquad (S2)$$

We prove this by 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 $k + 1$ messages:

$$\Box(|Y| = k \supset \Diamond|Y| > k)$$

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.

$$
\begin{array}{lll}
A_j \in \delta & \supset & A_j \in 7 \quad \text{(mrtl)} \\
& 3 & |Y| \geq j \quad \text{(R2)} \\
& \supset & M_j \in \beta \quad \text{(R1)} \\
& 3 & M_j \in \alpha \quad \text{(mtrl)} \\
& \supset & |X| \geq j \quad \text{(T1)} \\
& 3 & A_{j-1} \in \delta \quad \text{(T2)} \\
& 3 & A_{j-1} \in \gamma \quad \text{(mrtl)} \\
& 3 & |Y| \geq j-1 \quad \text{(R2)}
\end{array}
$$

This implies both hypotheses, that is,

$$A_j \in \delta \quad \supset \quad |X| \geq j$$

$$M_j \in \beta \supset |Y| \geq \mathrm{j} \quad - \quad 1$$

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

$$\forall\, j(A_j \in \delta \supset (uc(\alpha,\ M_{j+1})\ \vee \diamond(A_{j+1} \in \delta))) \qquad (T5')$$

$$\forall\, j(M_j \in \beta \supset (\diamond(|Y| \geq j) \wedge (uc(\gamma, A_j) \vee \diamond(M_{j+1} \in \beta)))) \qquad (R5')$$

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

$$uc(\gamma, A_k), \text{ or} \qquad (1)$$

$$\diamond M_{k+1} \in \beta. \qquad (2)$$

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

$$uc(\alpha, M_{k+1}), \text{ or} \qquad (1a)$$

$$\diamond A_{k+1} \in \delta. \qquad (1b)$$

NOW case la implies $\diamond M_{k+1} \in \beta$ (using mtr3), so case la 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 lb implies $\diamond |Y| \geq k+1$. This completes the proof of the system liveness property.

Conclusion

In addition to the protocol presented here, we have proved the correctness of three additional protocols using this approach: Stenning's full data transfer protocol [16], the alternating bit protocol (as stated by Bochmann [1]), and Brinch Hansen's multiprocessor network [3].

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

[1] Gregor V. Bochmann and Jan Gecsei. A unified method for the specification and verification of protocols. Proceedings of *IFIP* Congress 77, pages 229–234. North Holland Publishing Company, 1977.

[2] Per Brinch Hansen. Operating System Principles. Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[3] Per Brinch Hansen. Network: A multiprocessor program. *IEEE Transactions* on Software Engineering, SE-4 (3):194–199, May 1978.

[4] Dov Gabbay, Amir Pnueli, Sharon Shelah, Yonatan Stavi. On the temporal analysis of fairness. *Seventh* Annual ACM Symposium on Principles *of Programming Languages,* (Las Vegas) pages 163-173, Association for Computing Machinery, 28 January 1980.

[5] Donald I. Good and Richard M. Cohen. Principles of proving concurrent programs in Gypsy. Sixth Annual ACM Symposium *on Principles of Pro*gramming Languages, (San Antonio) pages 42-52, Association for Computing Machinery, 29 January 1979.

[6] Brent T. Hailpern. *Verifying Concurrent Processes Using* Temporal Logic. In preparation, PhD Thesis, Stanford University, 1980.

[7] C. A. R. Hoare. Monitors: An operating system structuring concept. Communications of the *ACM,* 17(10):549–557, October 1974.

[8] John H. Howard. Proving monitors. *Communications of the ACM,* 19(5):273–279, May 1976.

[9] Leslie Lamport. "Sometime" is sometimes "not never? On the temporal logic of programs. Seventh Annual *ACM Symposium* on *Principles* of Programming Languages, (Las Vegas) pages 174-185, Association for Computing Machinery, 28 January 1980.

[10] Leslie Lamport and Susan S. Owicki. Proving liveness properties of concurrent programs. In preparation, 1980.

[11] Susan S. Owicki. Specifications and proofs for abstract data types in concurrent programs. In F.L. Bauer and M. Broy, editors, Program Construction, pages 174-197. Springer-Verlag, 1979.

[12] Susan S. Owicki. Specification and verification of a network mail system. In F.L. Bauer and M. Broy, editors, Program Construction, pages 198-234. Springer-Verlag, 1979.

[13] Amir Pnueli. The temporal logic of programs. The 18th *Annual Symposium on Foundations of Computer* Science, (Providence, Rhode Island) pages 46-57, Institute of Electrical and Electronic Engineers, 31 October 1977.

[14] Amir Pnueli. A temporal semantics for concurrent programs. Unpublished, University of Pennsylvania, November 1977.

[15] Amir Pnueli. The temporal semantics of concurrent programs. *The* International Symposium on the Semantics of Concurrent Computation, (Evian) July 1979.

[16] Norman V. Stenning. A data transfer protocol. *Computer Networks,* **1(2):99–**110, September 1976.

[17] Carl A. Sunshine. Formal techniques for protocol specification and verification, *Computer,* **12 (9):20–27,** IEEE Computer Society, September 1979.

```
Transmitter ( x : unbounded input stream of message) : process

begin

    {initialize}
    WaitingForAck : = 1
    HighestSent := 0

    loop

        { send message }
        if HighestSent < WaitingForAck then
            HighestSent := HighestSent + 1
            data := read(X)
            mtr.send ( [ HighestSent, data ] )
            timer.start
        fi

        { service acknofedgements }
        if mrt.ExistsM then
            mrt.receive ( [ackno,ack] )
            if ackno = WaitingForAck then
                timer. cancel
                WaitingForAck := ackno + 1
            fi
        fi

        { service time-outs 3
        if timer. TimeoutsExist then
            timer.cancel
            mtr.send ( [HighestSent, data] )
            timer.8 tart
        fi

    end loop

end process
```

Figure 1

```
Receiver ( y : unbounded output stream of message) : process

begin

        {initialize 3
        NextRequired := 1

        loop

                { get message }
                mtr.receive ( [ messno, data] )

                if messno = NextRequired then
                        { service new message }
                        y := y @ data
                        NextRequired := NextRequired + 1
                fi

                { send acknowledgment 3
                mrt.send( [ NextRequired - 1, "ack" ] )

        end loop

end process
```

Figure 1

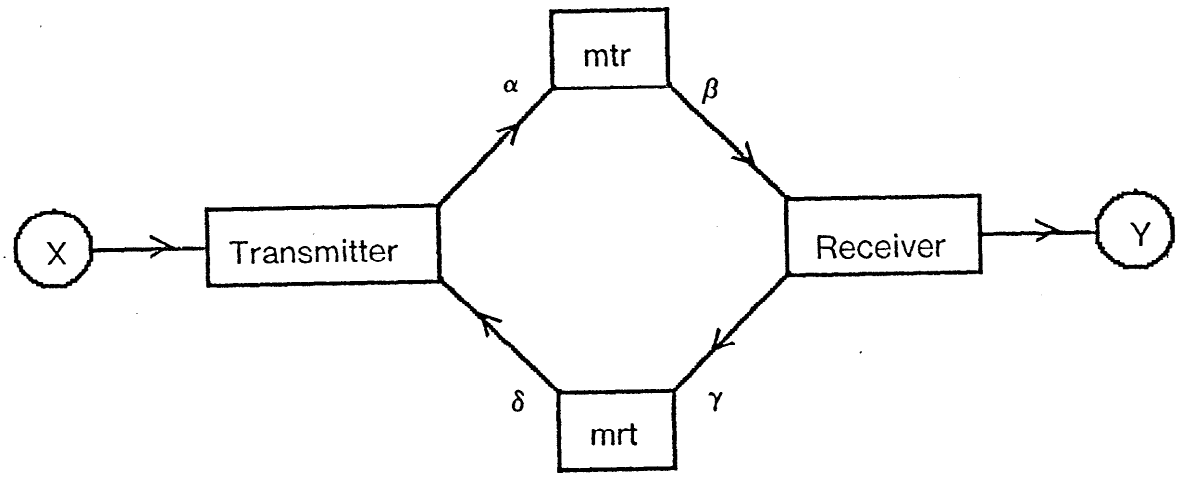

Figure 2