State Machines and Assertions: An Integrated Approach to Modeling and Verification of Distributed Systems

Thomas A. Joseph
Thomas Raeuchle
Sam Toueg*

TR 85-652
November 1984
(Revised May 1985 and November 1985)

Department of Computer Science
Cornell University
Ithaca, NY 14853

*This author was partially supported by the National Science Foundation under grant MCS 83-03135.

STATE MACHINES AND ASSERTIONS:
An Integrated Approach to Modeling and Verification of Distributed Systems

Thomas A. Joseph, Thomas Räuchle, Sam Toueg[1]

Department of Computer Science, Cornell University, Ithaca, NY 14853.

[1] This author was partially supported by the National Science Foundation under grant MCS 83 - 03135.

ABSTRACT

This paper describes a methodology for modeling and verifying protocols for asynchronous message passing systems. It combines the techniques of finite state analysis and axiomatic verification. It overcomes the problem of state explosion by using variables and logical assertions where the finite state approach would require a large number of states. By explicitly including states where interactions between processes occur, the complexity of assertional proofs is significantly reduced. Properties like freedom from deadlock, freedom from unspecified message receptions, boundedness of channel size, and partial correctness can be proved. Properties of channels like losing or garbling messages can be modeled, as can premature and non-premature timeouts. The technique is illustrated by proving a sliding window flow control protocol and an alternating bit protocol that is correct only if timeouts are non-premature.

1. Introduction

Two basic directions have been followed in proving properties of asynchronous message passing protocols: the finite state machine approach and the axiomatic approach. In the finite state machine approach, each participating process is modeled by a finite state machine, and the communication links are modeled by channels. To prove a property of a protocol, one constructs a global state machine, whose states are members of the Cartesian product of the states of the individual state machines and of the channels. The problem of verifying a protocol reduces to determining whether certain global states are reachable from the initial configuration. Properties that can be verified in this way include freedom from deadlock, freedom from unspecified message receptions, and boundedness of channels [Boc78, Sun79, Boc80, Gou82, Bra83].

A limitation of this method is that a separate state is required for each possible value of a variable and for each possible channel configuration. This makes the size of the global state transition graph exponential in the range of variable values. If the channels are unbounded, this problem is undecidable. In [Rae85] it is shown that even when the channel size is bounded, the problem of determining whether a deadlock state is reachable is PSPACE hard if the channels are FIFO, and NP hard if not.

The axiomatic approach stems from the work of Floyd [Flo67], who used predicate calculus assertions to describe the semantics of programs. The same approach is used in [Hoa69] and [Dij76]. Ashcroft and Manna [Ash71] extended this idea to parallel processes by showing how to transform such a system to an equivalent non-deterministic program. The resulting programs and their proofs could be large and complex, and Ashcroft and Manna presented a method of simplifying them based on the independence of parallel branches. Owicki and Gries [Owi76] developed a proof technique in which a property is proved for each process in isolation, then interference freedom is shown. The complexity of proofs in both methods is similar: as Lamport observed in [Apt80], "the same amount of verification is required in both methods". Schlichting and Schneider [Sch84] extended the method in [Owi76] to asynchronous message passing systems, modeling communication links by shared auxiliary variables, and introducing proof rules for sending and receiving messages. The methods above were designed for proving general parallel programs. In such programs, the execution of any statement in one process could interfere with the proof of another. As a result, they require a large number of interference freedom proofs.

In this paper, we take advantage of the fact that in message passing systems with no shared memory, the points where interference could occur are limited to the receipt of messages. We use concepts from the finite state approach to explicitly identify states where interference could occur. This simplifies proving properties of asynchronous message passing protocols by reducing the burden of proving interference freedom. At the same time, the method overcomes the state explosion of the finite state approach by modeling variables explicitly and using assertions to describe their properties. The model is flexible enough to allow a protocol to be specified and verified by either the axiomatic or the finite state method, but its advantage is that it permits a combination of both.

We also present a comprehensive model for timeouts, which is powerful enough to express non-premature timeouts. We demonstrate the use of this model in the proof of an alternating bit protocol that is correct only if timeouts are non-premature.

The paper is organized as follows. In Section 2 we describe the basic model, and in Section 3 we present the proof system. Section 4 completes the model to allow for message loss and timeouts. A discussion of our results concludes the paper.

2. The Basic Model

The systems we model consist of a set of processes exchanging messages across an asynchronous communication network. Processes perform local computations and have no shared memory. Different properties of communications networks - FIFO message delivery, unreliability, bounded network capacity, etc. - can be modeled. The model can be applied to high-level application programs as well as to low-level data link protocols.

2.1. Processes

A process is modeled by a set of *local variables* and a *state transition graph* - a directed graph, with a finite set S of nodes, called *states*, and a set T of labeled edges, called *transitions*, between nodes in S. A process starts executing in its *initial state* and performs transitions non-deterministically.

If a process enters a state out of which there are no transitions (*final state*), it terminates.

There are two types of transitions: *send* and *receive* transitions (Fig. 1). The send transition in Figure 1(a) is *enabled* if the process is in state S_1 and the boolean *guard* B is true. The receive transition in Figure 1(b) is enabled if the process is in state S_1, the guard B is true, and the receive statement is executable (see Section 2.2). A process can only execute enabled transitions. A transition from S_1 to S_2 is executed by carrying out the local computation LC_1, executing the send or receive statement as described below, and performing the local computation LC_2. Guards and local computations involve only local variables of the process. If the guard is the value **true** or if a local computation is absent, it is not included in our figures.

A given process can be represented in our system in more than one way. It is possible to model it using a single state and guarded transitions from this state back to itself. This is equivalent to the axiomatic approach. On the other hand, local variables can be eliminated by introducing a state for each value a variable takes. This results in a finite state machine. The strength of our technique is that a protocol designer can combine the two

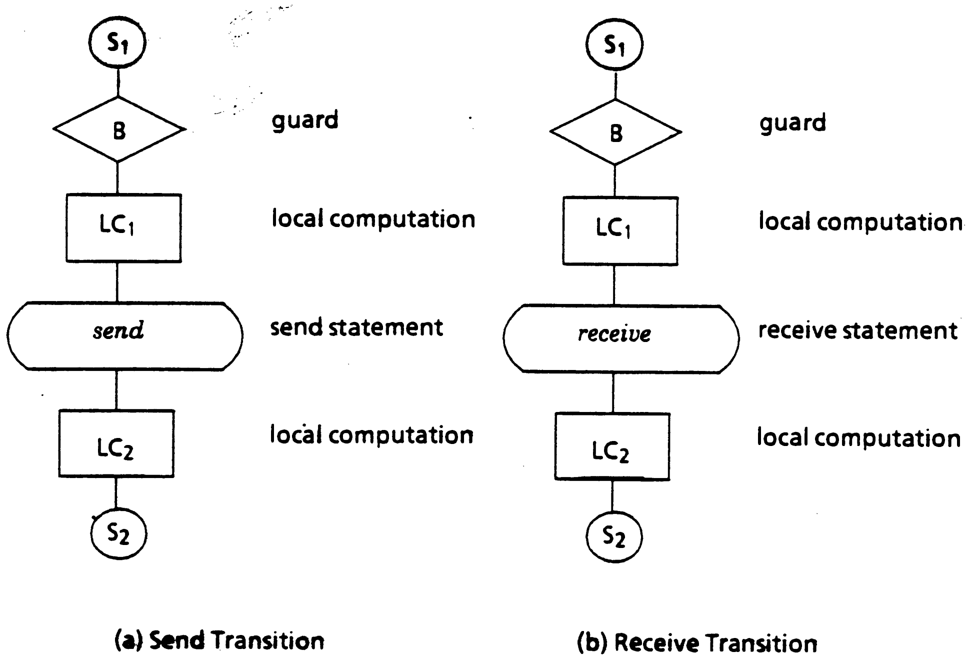


Figure 1.

approaches, choosing states and variables in a way that reflects his or her intuitive understanding of the system behavior. For example, a state would be chosen to represent the fact that all buffers are full, or that a process is blocked waiting for a response. Properties of states can then be conveniently described by assertions as shown later.

2.2. Channels

A *channel* is a named unidirectional link between two processes, across which a *source* sends *messages* to a *destination*. A message is a tuple of *data items* contained in named *data fields*. Messages have types; the type of a message m is denoted $m.type$. A channel c is represented by two *channel sequences* and a *successor function*. $Sent_c$ is the sequence of messages placed on c by the source, and $Received_c$ is the sequence of messages received on c by the destination. The successor function, $next_c$, depends on the values of the channel sequences and on properties of the communication medium modeled. Its value is the smallest set S of messages such that the next message that can be received on channel c belongs to S. For example, for a non-FIFO, loss-free channel that does not generate spurious messages, $next_c$ is the set $\{Sent_c$ - $Received_c\}$, since any message sent but not yet received could be the next one received. If the channel is FIFO, $next_c$ is $\{Sent_c.[\text{length}(Received_c) + 1]\}$, where $X.i$ is the ith message in sequence X. For a FIFO channel on which messages could be garbled but not lost, $next_c$ is $\{Sent_c.[\text{length}(Received_c) + 1]\} \bigcup \{<garbage>\}$, where $<garbage>$ is a garbled message. A channel is said to be *empty* if the value of $next_c$ is the empty set. We let $empty(c)$ stand for the assertion "channel c is empty".

A send statement has the form

$$\textbf{send } (c,\ type{:}t,\ f_1{:}e_1,\ f_2{:}e_2,\ \cdots,\ f_k{:}e_k)$$

and the following semantics : in channel c, place a message of type t, with data fields named $f_1, f_2, \cdots, f_k$. The values placed in these fields are the values of $e_1, e_2, \cdots, e_k$, which are expressions involving local variables of the process executing the send. Executing a send statement extends the sequence $Sent_c$ by the message sent.

A receive statement has the form

$$\textbf{receive } (c,\ type{:}t,\ f_1{:}v_1,\ f_2{:}v_2,\ \cdots,\ f_k{:}v_k).$$

Here $v_1, v_2, \cdots, v_k$ are local variables. This statement has the following meaning. If channel c is not empty, and the message at the head of c (an element of $next_c$) is of type t, remove the message from c and set the values of variables $v_1, v_2, \cdots, v_k$ to the values in the fields $f_1, f_2, \cdots, f_k$ of the message. This statement is *executable* only if the channel c is not empty, and the message at the head of c is of type t. Executing a receive statement extends $Received_c$ by the message received.

2.3. Example

Figure 2 shows a sliding window flow control protocol, as modeled in our system. It is similar to a *selective repeat* protocol described in [Tan81]. There are two processes, the sender and the receiver, connected by two perfect FIFO channels - SR from the sender to the receiver, and RS from the receiver to the sender. The sender takes data from the array *info* and sends them in numbered messages to the receiver on channel SR. Sequence numbers are in the range 0 to $M-1$. The maximum window size for the sender is SW_SIZE, that is, it does not transmit more than $SW\_SIZE - 1$ new messages after the first unacknowledged one. The receiver window size is RW_SIZE. It accepts a message if its sequence number is in the receive window, and rejects it otherwise. If it accepts a message, it places the data into the array *accepted* and sends an acknowledgement on channel RS, containing the sequence number of the message accepted. Otherwise, it sends a negative acknowledgement containing the sequence number of the message rejected. On receiving a negative acknowledgement, the sender retransmits the rejected message. To be able to reuse sequence numbers without ambiguity, SW_SIZE is less than or equal to $M - 1$.

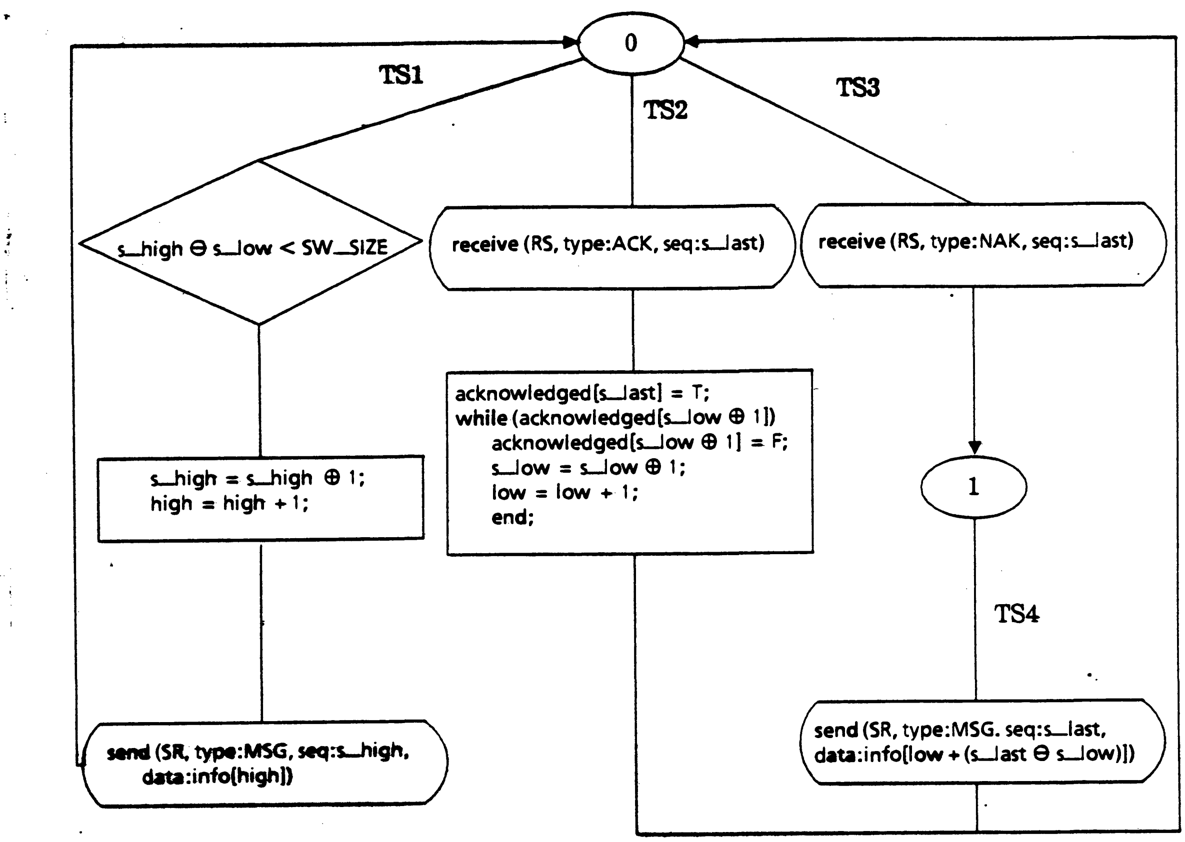
3. Proof System

To prove a property of a distributed system modeled as above, a *global state transition graph* is first constructed. Its states are members of the Cartesian product of the states of the individual processes. $S = (s_k, s_l, \cdots, s_m)$ denotes a global state, where process 1 is in its kth state, process 2 is in its lth state, and so on. $S_i.j$ denotes the jth component of global state S_i. Thus in the example above, $S.1$ is s_k. Each global state transition corresponds to a state transition in one, and only one, of the component processes.[2] The global state transition graph contains a transition T from state S_i to state S_j if, and only if, there is an n such that there is a transition T from state $S_i.n$ to $S_j.n$ in the state transition graph of process n, and for all $k \neq n$, $S_i.k = S_j.k$. Note that the size of the global transition graph is independent of the channel size, and proportional to the product of the sizes of the graphs of the component processes. Figure 3 shows the state transition graphs for the processes in Figure 2 and the system state transition graph.

The next step is to associate a predicate with each global state. Predicates are assertions on the values of the local variables of the processes and the values of the channel sequences. The predicate associated with state S_i

---

[2] This does not allow synchronous message passing to be modeled. By allowing a global state transition to correspond to a local transition in both a source and a destination process, and making appropriate changes in the proof rules. the model can be extended to cover synchronous message passing.



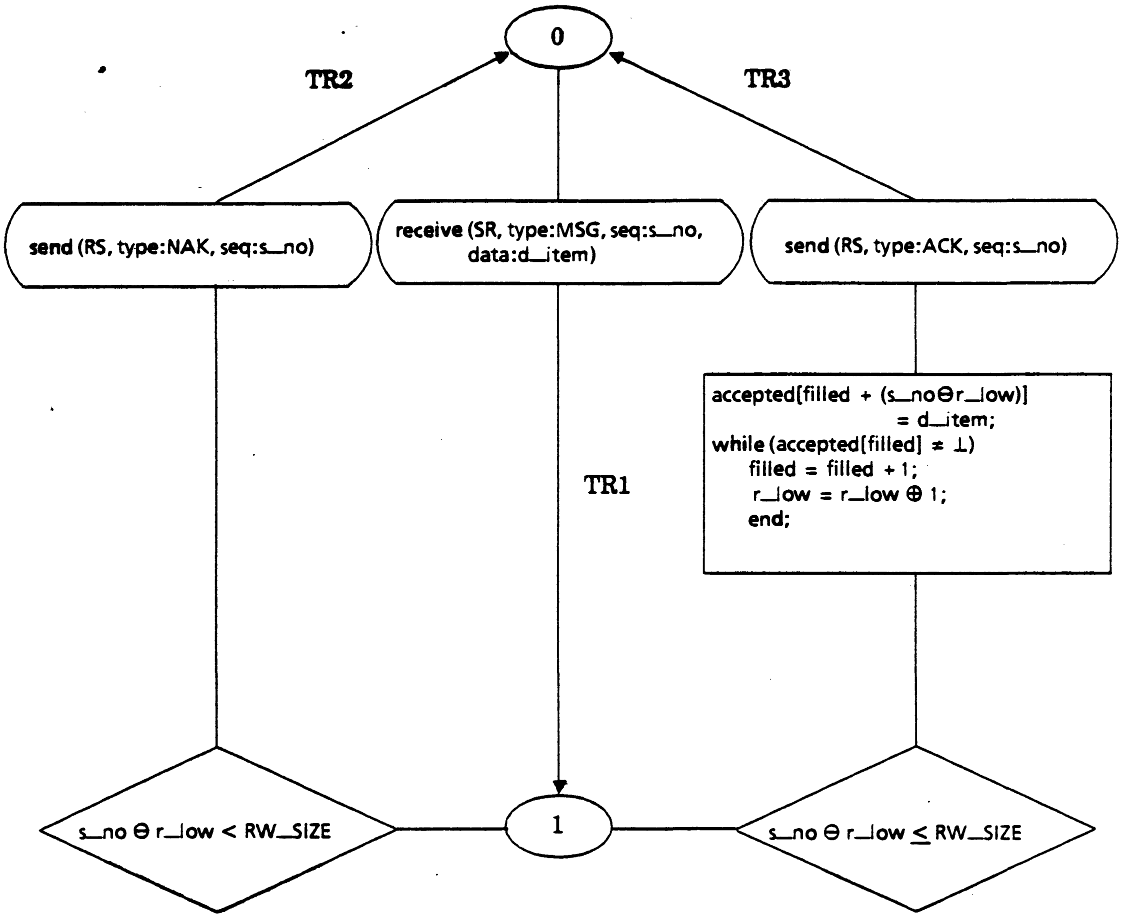
Notation

$\oplus$ ($\ominus$)	: modulo M addition (subtraction)
[a ... b]	: the interval a, a $\oplus$ 1, a $\oplus$ 2, ... , b
(a ... b)	: the interval a $\oplus$ 1, ... , b $\ominus$ 1
(a ... b]	: the interval a $\oplus$ 1, ... , b $\ominus$ 1, b

Sender variables

info[1..]	: array of data items to be transmitted to receiver
s_high	: (Modulo M variable) sequence number of the last message sent, ignoring retransmissions (initialized to 0).
s_low	: (Modulo M variable) sequence number of the message before the first unacknowledged message (initialized to 0). ((s_low ... s_high] is the sending window)
s_last	: (Modulo M variable) sequence number of last message for which a positive or a negative acknowledgement has been received.
info[high]	: data item sent in message with sequence number s_high (high is initialized to 0).
info[low]	: data item in the message with sequence number s_low (low is initialized to 0).
acknowledged[i]	: (Boolean) *T* only if i $\in$ (s_low ... s_high] and the sender has received an acknowledgement for the last message sent with sequence number i. (Since (s_low ... s_high] is empty on initialization, acknowledged[i] is initialized to *F* for all i.)
SW_SIZE	: maximum sending window size (SW_SIZE $\leq$ M - 1).

Figure 2a. Sender Protocol and Variables.



Receiver variables

accepted[1..]	: array into which data items accepted by the receiver are placed (each element is initialized to *undefined*).
filled	: all items in accepted[1-filled] have been received and acknowledged. No array element in this range is *undefined*.(initialized to 0)
r_low	: sequence number of message that carried the data item in accepted[filled]. (r_low, initialized to 0, is the lower bound of the receiver's window)
s_no	: sequence number of last message received.
d_item	: data item in last message received.

Figure 2b. Receiver Protocol and Variables

will be denoted by P_i. It must then be shown that the predicate associated with the initial global state is satisfied by the initial values of the variables, with the channel sequences empty. Also, for every transition S_i ----- T -----> S_j, $\{P_i\}$ T $\{P_j\}$ must be a theorem in our proof system, where P_i is the predicate associated with S_i.

Our proof system consists of the axioms and inference rules of a programming logic system such as in [Sch84]. The axioms for the **send** and **receive** statements follow.

Sending a message m on a channel c does not change the value of any variable of the source; it only affects the state of c, extending $Sent_c$ by m. Let $P^{x_1, \cdots, x_k}_{y_1, \cdots, y_k}$ represent the assertion P with $y_1, \cdots, y_k$ textually substituted for $x_1, \cdots, x_k$, and let $X \bullet R$ be the sequence formed by concatenating record R onto the sequence of records X. The send axiom is as follows.

$$\{ P^{Sent_c}_{Sent_c \bullet \{type:t,\ f_1:e_1, \cdots, f_k:e_k\}} \}$$

$$\textbf{send}\ (c,\ type{:}t,\ f_1{:}e_1, \cdots, f_k{:}e_k)$$

$$\{ P \}$$

The receive statement, on the other hand, could alter values of destination's variables. It also changes the state of the channel. The receive axiom is

$$\{ \forall m \in next_c \text{ with } m.type = t : P^{Received_c,\ v_1, \cdots, v_k}_{Received_c \bullet m,\ m.f_1, \cdots, m.f_k} \}$$

$$\textbf{receive}\ (c,\ type{:}t,\ f_1{:}v_1, \cdots, f_k{:}v_k)$$

$$\{ P \}$$

The inference rule for the send (receive) transition is the following. Let T be the transition S_i -- B -- C_1 -- **send (receive)** -- C_2 ---> S_j.

Send (Receive) rule

$$\frac{\{P \wedge B\}\ C_1\ \{R\} \qquad \{R\}\ \textbf{send (receive)}\ \{S\} \qquad \{S\}\ C_2\ \{Q\}}{\{P\}\ T\ \{Q\}}$$

In Appendix A we show the assertions associated with each of the global states of Figure 3 and give a detailed proof of the theorem $\{P_{0,0}\}$ $TS1$ $\{P_{0,0}\}$ corresponding to the transition $TS1$ from state $S_{0,0}$ back to $S_{0,0}$. The proofs for the other transitions are similar.

Finally, we must show that the properties desired of the system follow from the predicates associated with the global states. Among the properties

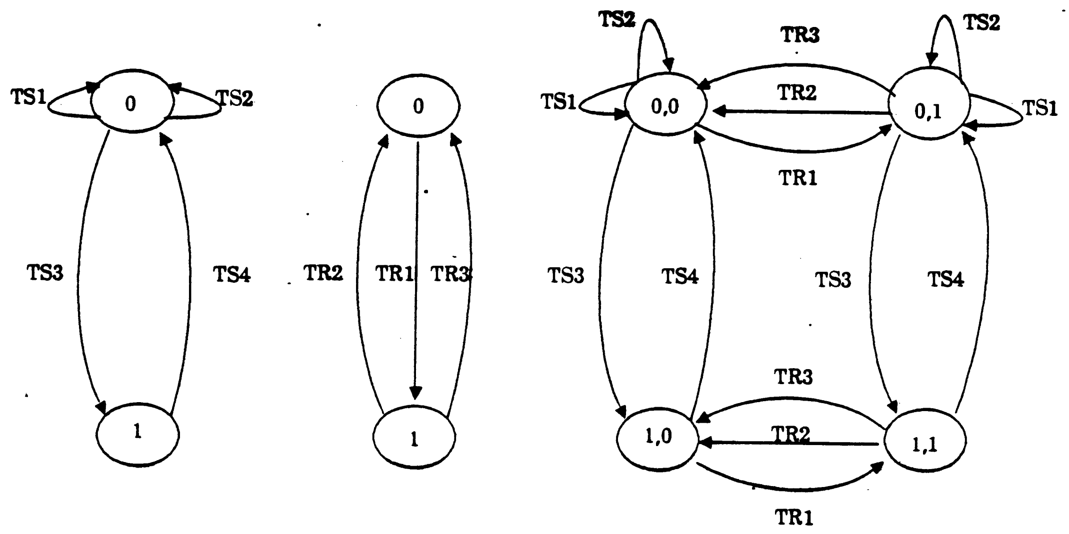


Figure 3. State transition graph for sliding window protocol.

that can be proved in our system are freedom from unspecified message receptions, freedom from deadlock, and various safety properties.

3.1. Freedom from unspecified message receptions

Unspecified message receptions can occur in protocols for which incoming messages are not buffered automatically, such as data link protocols. In this case, whenever a channel is not empty, a destination must be in a state from which it is possible to execute a receive transition for the type of message at the head of the channel. Otherwise, the message would be lost and the message reception is said to be unspecified. To prove that a protocol is free from unspecified receptions, one shows that for every state S_k, the predicate P_k implies that if $next_c$ contains a message of type t, then there is at least one enabled receive transition of type t on channel c out of S_k. This may be expressed formally as follows.

Let the set N_c contain the types of the messages in $next_c$. For each type t in N_c, let $B_i(0<i\leq j)$ be the guards on the receive transitions of type t on channel c. Then the protocol is free from unspecified receptions if, for all states S_k with assertion P_k,

$$P_k \Rightarrow \forall \text{ channels } c : \{\forall \text{ types } t \in N_c : [\exists\, i : B_i]\} \qquad \textbf{(3.1)}$$

3.2. Freedom from Deadlock

A protocol normally terminates by entering a final state. It also terminates if the system enters a non-final state out of which there is no enabled transition. This form of termination is called *deadlock* and is usually undesirable. To prove that a protocol is free from deadlock, we need to show that for every non-final state S_k, the predicate P_i implies that either there is an enabled send transition out of S_k, or that a channel contains a message for which there is an enabled receive transition. If we have proved that the protocol is free from unspecified receptions, there is always an enabled receive transition for the next message on the channel. Thus, a deadlock occurs only if the system is in a non-final state from which there is no enabled send transition and all the channels are empty. Formally,

Let N_c be the set of message types of the messages in $next_c$. For each message type t in N_c, let $B_i(0<i\leq j)$ be the guards on the receive transitions of type t on channel c out of S_k. (If there is no outgoing transition for a type t, the corresponding guard is **false**.) Let $G_l(0<l\leq m)$ be the guards on the send transitions. The protocol is free from deadlock if for all non-final states S_k,

$$P_k \Rightarrow (\exists l : G_l) \vee \exists \text{ channel } c : (\exists \text{ type } t \in N_c : [\exists i : B_i]) \qquad (3.2a)$$

If the protocol is free from unspecified receptions, (3.1) holds, and (3.2a) reduces to

$$P_k \Rightarrow (\exists l : G_l) \vee (\exists \text{ channel } c : c \text{ is not empty}) \qquad (3.2b)$$

3.3. Boundedness of channel size

The number of messages in a channel that does not generate messages is bounded by length($Sent_c$) - length($Received_c$). To prove that it is bounded by M, one proves that every predicate P_k implies that length($Sent_c$) - length($Received_c$) $\leq M$.

3.4. Other safety properties

In general, any safety property P can be proved by showing that every predicate P_k implies P.

3.5. Example

In Appendix B, we show that the protocol in Figure 2 is free from deadlock, that the *accepted* sequence is an initial subsequence of the *data* sequence, and that the number of messages in any channel is always less than M.

4. The Complete Model

In this section, we extend the basic model to cover loss of messages and timeouts. As in [Cho83], this is achieved by introducing special state transitions into the model. In contrast with [Sha83], we express timeout behavior without modeling real time explicitly, but by considering the restrictions on the ordering of events imposed by timeout semantics.

4.1. Lost messages

Loss of messages can be modeled as a property of a channel c by suitably defining the function $next_c$. However, loss-free channels are easier to reason about, and so we present an alternative method of modeling lost messages. A lossy channel can be modeled by treating the channel itself as loss-free, and adding transitions to the state transition diagram of the destination that allow it to (non-deterministically) receive a message and return to the same state without taking any action on that message. In other words, the destination is viewed as actually receiving the message on a loss-free channel and simply discarding it. Any property proved using this state transition diagram, assuming the channels are loss-free, is true of the original protocol, with lossy channels.

"Dropping" a message can be modeled by adding receive transitions as described above. To improve readability, we denote these as *lose transitions*. Lossy channels are then modeled as follows. For every state S from which there is a transition

$$S---B---C_1---\mathbf{receive}(c,\ type\colon t, \cdots)---C_2--\rightarrow T,$$

we add a lose transition,

$$S---B---\mathbf{lose}(c,\ type\colon t,\ \cdots)--\rightarrow S.$$

Like the receive statement, the lose statement chooses a message m non-deterministically from $next_c$, and extends the sequence $Received_c$ by that message. The proof rule for the **lose** statement is similar to that for the **receive**:

$$\{\ \forall m \in next_c \text{ with } m.type = t : P^{Received_c,\ v_1,\ \cdots,\ v_k}_{Received_c \bullet m,\ m.f_1,\ \cdots,\ m.f_k}\ \}$$

$$\mathbf{lose}(c,\ type : t, f_1 : v_1, \cdots, f_k : v_k)$$

$$\{\ P\ \}$$

4.2. Timeouts

Many protocols use timeouts to detect lost messages. We call a process that sends a message and sets a timer the *sender*, and the process that receives the message and sends an acknowledgement the *receiver*. For every message sent, there is an associated *timer-ID* that identifies a particular software or hardware timer. This timer is *set* to a *timeout interval* when the message is sent. When an acknowledgement for a message is received, the timer with the associated timer-ID is *canceled.* A timer that is not canceled *expires* at the end of its timeout interval and triggers an interrupt. If timer interrupts are not masked, the sender processes the timeout immediately. Otherwise, the timeout remains pending until interrupts are unmasked.

To model timeouts we introduce *timer types.* A timer type is a set of timer-ID's. Timers are manipulated using the **set** statement, which adds a timer-ID to a timer type, and the **cancel** statement, which removes a timer-ID from a timer type, if it contains the timer-ID, and leaves the timer type unchanged otherwise. The proof rules are:

$$\{\, P^t_{t \cup \{i\}} \,\} \qquad\qquad \{\, (P \wedge i \notin t) \vee (P^t_{t - \{i\}}) \,\}$$

$$\mathbf{set}(type\colon t,\ id\colon i) \qquad\qquad \mathbf{cancel}(type\colon t,\ id\colon i)$$

$$\{\, P \,\} \qquad\qquad \{\, P \,\}$$

Finally, we include a *timeout transition,* which has the following form:

$$S_1 \ --- \ \mathbf{timeout}(type\colon t,\ id\colon x) \ --\!\!\rightarrow S_2$$

This transition is executable only if the process is in state S_1 and t is not empty. The transition above is performed by non-deterministically removing a timer-ID from t, assigning its value to the variable x, and entering the state S_2. The proof rule for the **timeout** statement is

$$\{\, \forall\, i : i \in t : P^{t,x}_{t - \{i\},\, i} \,\}$$

$$\mathbf{timeout}(type\colon t,\ id\colon x)$$

$$\{\, P \,\}$$

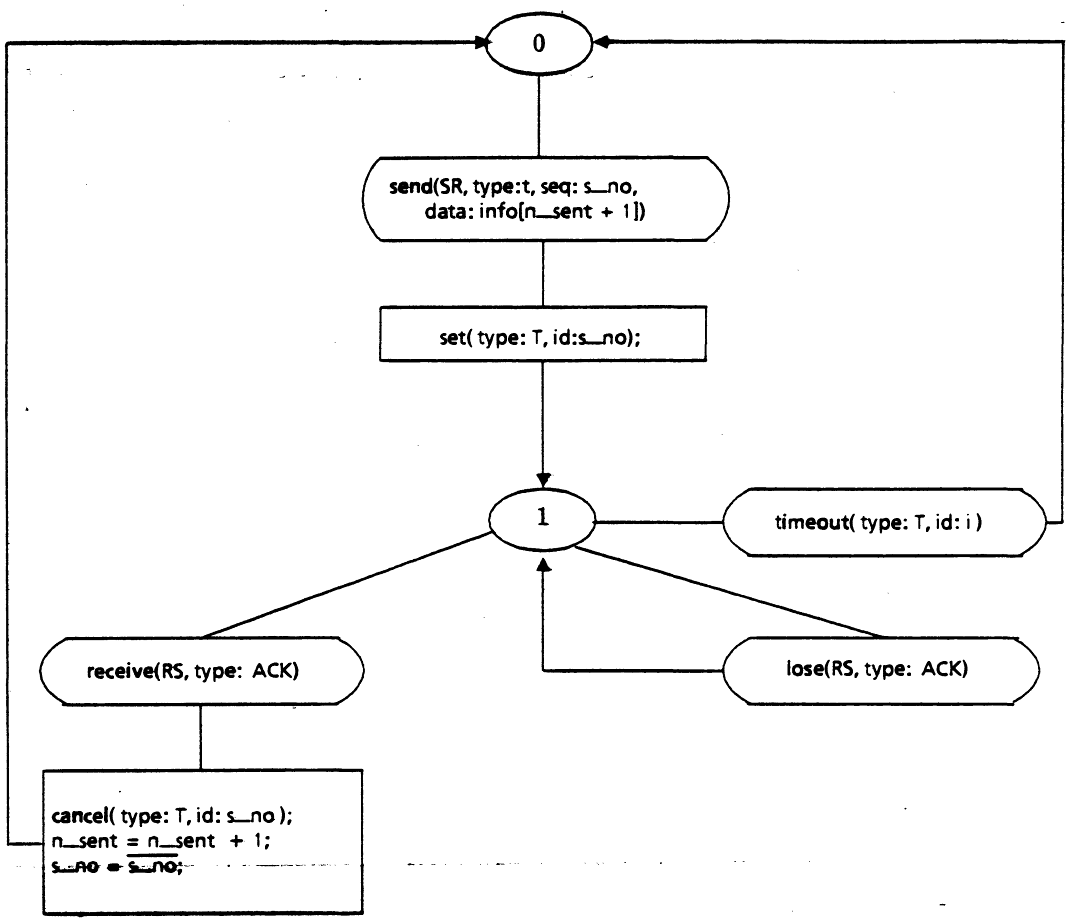
Timeouts are modeled as follows. A **set** statement is executed whenever the protocol sets a timer; a **cancel** statement whenever it cancels one. From every sender state S where timeout interrupts are not masked, the designer includes a transition

$$S \ --- \ \mathbf{timeout}(type\colon t,\ id\colon x) \ --\!\!\rightarrow T\ ,$$

where T is the state entered by the protocol when a timeout interrupt occurs. The sender may perform timeout transitions non-deterministically in any state where timeouts are not masked; this models the asynchronous nature of timeouts.

When timeout transitions are included, a deadlock state is a non-final one out of which there is no enabled send, receive or *timeout* transition. The condition for freedom from deadlock (Sec. 3.2, Condition 3.2b) becomes

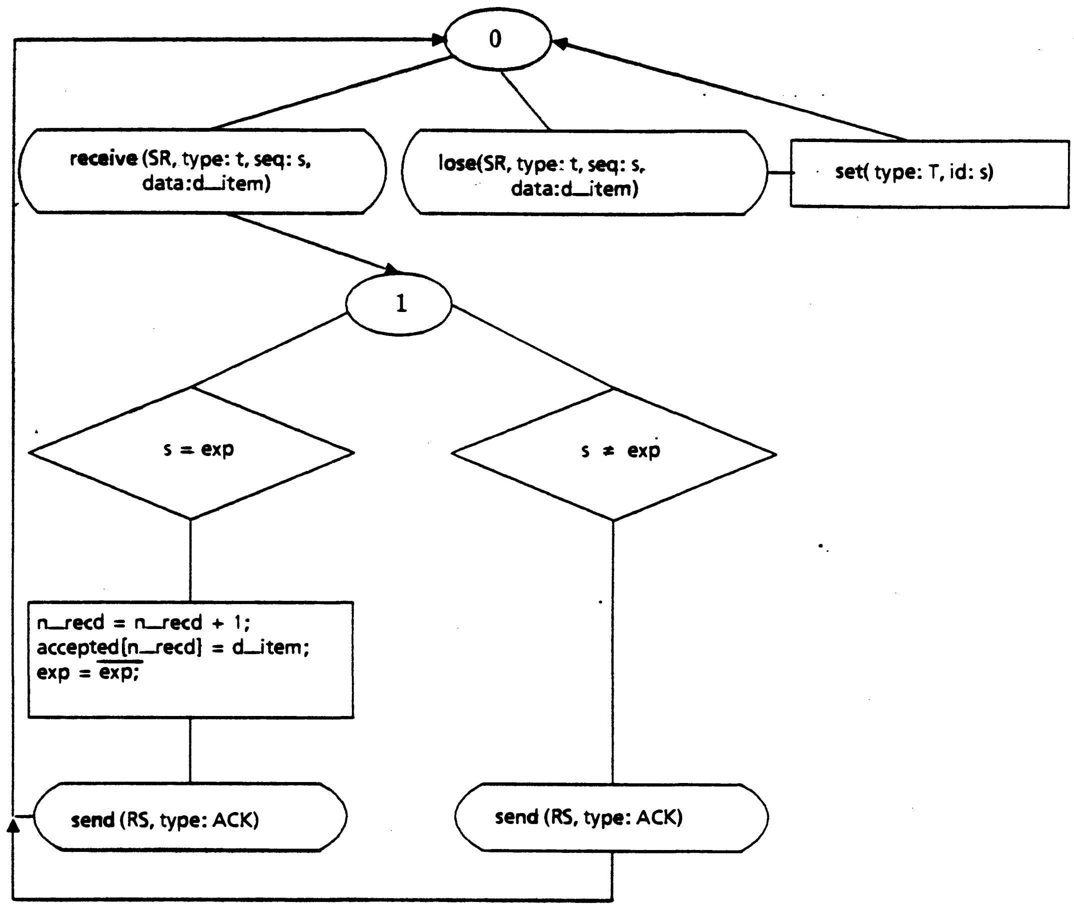
> Let N_c be the set of message types of the messages in $next_c$. For each message type t in N_c, let $B_i(\ 0<i\leq j\ )$ be the guards on the receive transitions of type t on channel c out of S_k. (If there is no outgoing transition for a type t, the corresponding guard is **false**.) Let $G_l(\ 0<l\leq m\ )$ be



Sender variables

info[1..]	: infinite array of data items to be transmitted
n_sent	: number of data items sent (initialized to 0)
s_no	: sequence number of message to be sent

Figure 4a. Sender in Alternating Bit Protocol



Receiver variables

accepted[1..]	: array to place data items accepted (all elements are initialized to *undefined*)
d_item	: data item in last message received
s	: sequence number of last message received
exp	: sequence number of next message expected (initialized to 0)
n_recd	: number of data items accepted (initialized to 0)

Figure 4b. Receiver in Alternating Bit Protocol

the guards on the send transitions. Let TT be the set of timer types for which there is an outgoing timeout transition. The protocol is free from deadlock if for all non-final states S_k,

$$P_k \Rightarrow (\ \exists l : G_l\ ) \vee \exists \text{ channel } c :\ (\ \exists \text{ type } t \in N_c : [\exists i : B_i]\ ) \vee \quad (4.1a)$$

$$(\ \exists \text{ timer type } T \in TT : T \neq \emptyset\ )$$

If the protocol is free from unspecified receptions, (4.1a) reduces to

$$P_k \Rightarrow (\ \exists l : G_l\ ) \vee\ (\ \exists \text{ channel } c : c \text{ is not empty} \vee \quad (4.1b)$$

$$(\exists \text{ timer type } T \in TT : T \neq \emptyset)$$

This method of modeling timeouts is adequate for a system in which *premature timeouts* could occur, that is, an acknowledgement could be received for a message after its timer expires. The scheme described above does not model a system in which only *non-premature* timeouts occur, because it is always possible for the sender to take a timeout transition, even if the corresponding message or its acknowledgement has not been lost.

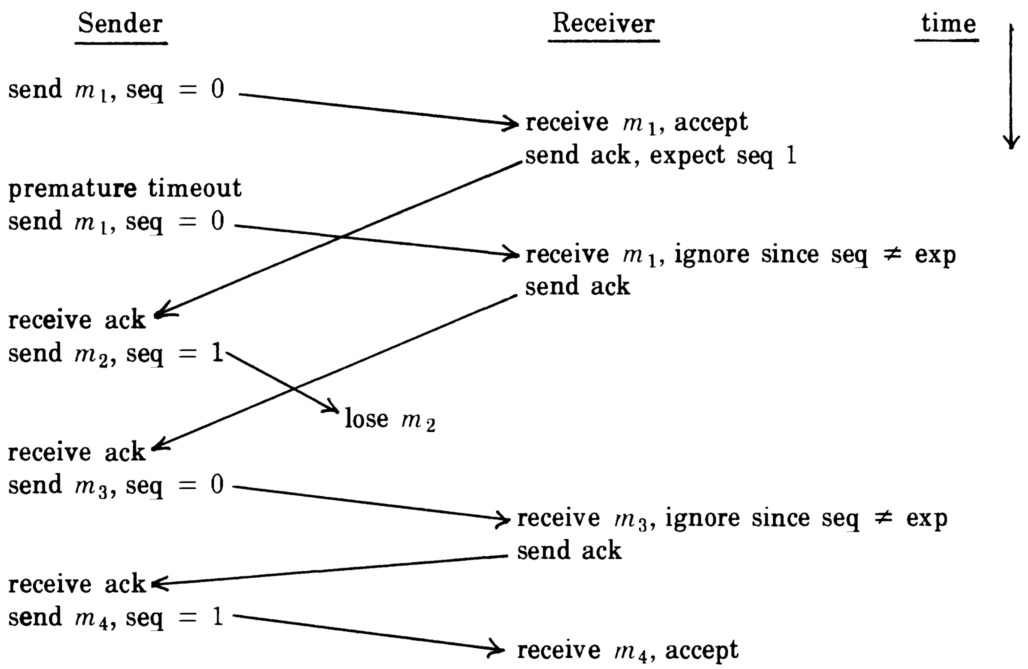


Figure 5. Failure of Alternating Bit Protocol with premature timeouts

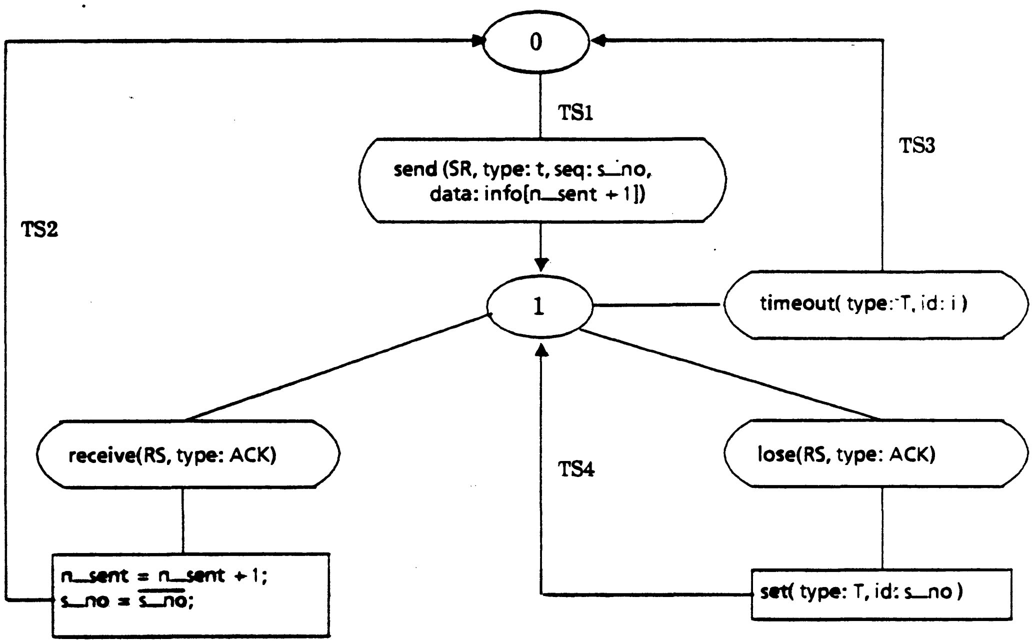


Figure 6a. Sender Protocol with non-premature timeouts

4.3. Example

As an example, we present a variant of the alternating bit protocol. In this protocol, the sender transmits messages with a one-bit sequence number, and waits for an (unnumbered) acknowledgement from the receiver. When the sender receives an acknowledgement, it transmits the next message, giving it a sequence number equal to the complement of that of the last message sent. If a timeout occurs, the sender retransmits the last message, giving it the same sequence number as before. If the receiver receives a message with the same sequence number as the last one accepted, it sends an acknowledgement to the sender, but does not accept the message. Otherwise, it accepts the new message and sends back an acknowledgement. Figure 4 depicts this protocol as modeled with lost messages and timeouts.

The protocol described above is correct only if all timeouts are non-premature. The scenario in Figure 5 demonstrates how it could fail in the presence of premature timeouts—the receiver accepts message m_4 without ever accepting m_3 and m_2. Such timing problems motivate the need for modeling non-premature timeouts. This is described in the next section.

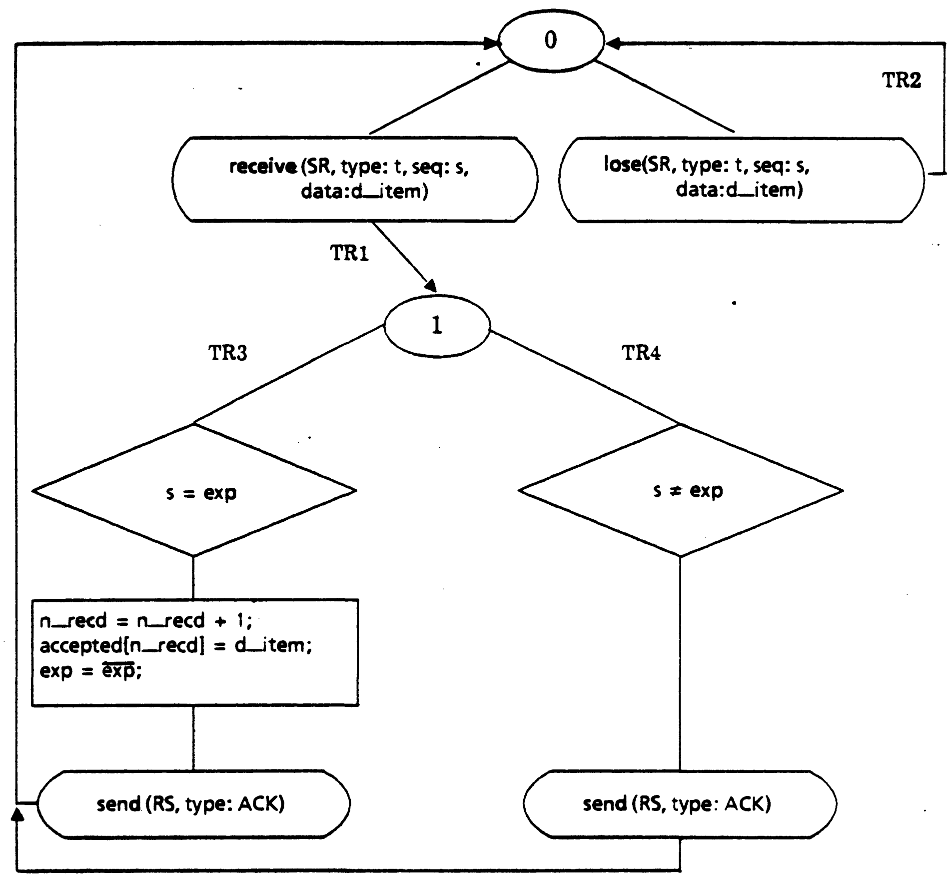


Figure 6b. Receiver Protocol with non-premature timeouts

4.4. Non-premature timeouts

The basic property of a non-premature timeout is that a timeout for a message occurs only if the message or its acknowledgement is lost. This is modeled as follows. Instead of executing a **set** statement at the time a message is sent, it is executed at the time when the message or its acknowledgement is lost. Hence **canceling** of timeouts is not necessary. When a message is lost, the corresponding lose transition in the receiver is followed by a **set** statement for the timer of the message lost. When an acknowledgement is lost, the sender executes **set** statements for every message that would have been acknowledged by it. (An acknowledgement may acknowledge more than one message.) This scheme ensures that a timer-ID is added to a timer type if, and only if, a message or its acknowledgement is lost. Hence, a timeout transition corresponding to a message can be taken only under these conditions, and the basic property of non-premature timeouts is modeled.

Figure 6 illustrates the alternating bit protocol as modeled with non-premature timeouts.

4.5. Proof of the alternating bit protocol

In this section we use our model of non-premature timeouts to prove that if all timeouts are non-premature, the sequence of accepted messages in our variant of the alternating bit protocol is an initial subsequence of the messages sent. Because this protocol could fail if timeouts are not premature, this proof demonstrates the power of our model of non-premature timeouts. We also prove that the protocol is free from deadlock and unspecified message receptions, and that the channel size is bounded by 1.

Figure 7 depicts the system state transition diagram and Figure 8 shows $P_{i,j}$ for all states $S_{i,j}$. For all i and j, $P_{i,j}$ implies RES, and RES implies that the accepted sequence of messages is an initial subsequence of *info*.

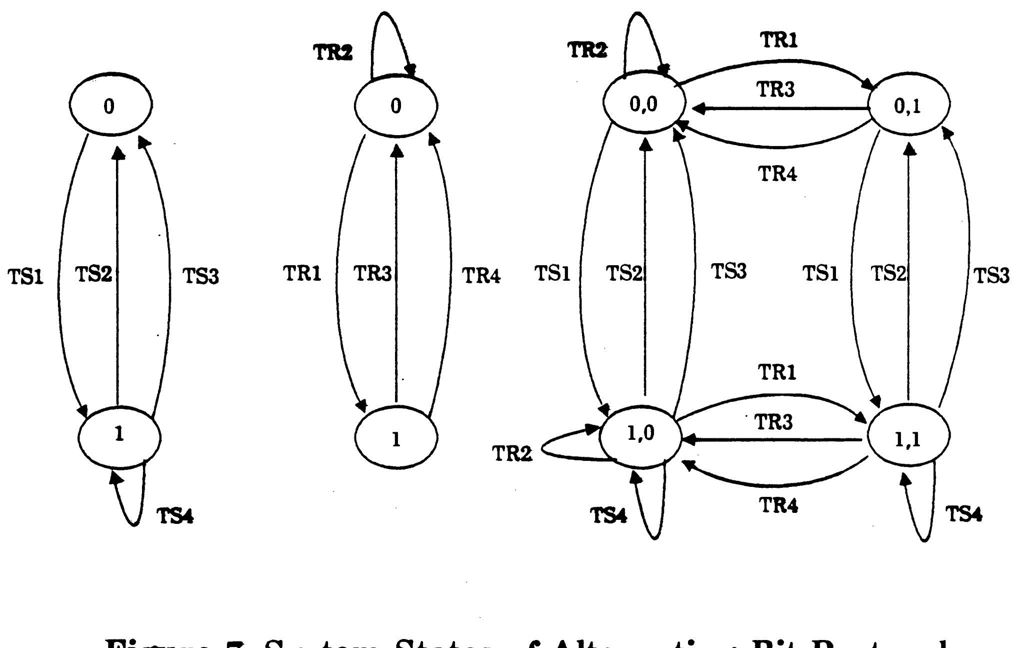


Figure 7. System States of Alternating Bit Protocol

State $S_{1,0}$ is the only state in which a deadlock could occur, since all other states have enabled send transitions out. It follows from $P_{1,0}$ that the assertion

$$size\_of(SR) + size\_of(RS) + |T| = 1 \qquad (4.2)$$

is true in $S_{1,0}$. (4.2) implies that one of the transitions out of state $S_{1,0}$ is always enabled and no deadlock can occur.

State $S_{1,0}$ is the only state in which unspecified message receptions could occur, since for all other states, $P_{i,j}$ implies that

$$size\_of(SR) + size\_of(RS) = 0$$

i.e., both channels are empty. It can be seen from Figure 7 that there is an enabled receive transition for both types of messages t and ACK out of state $S_{1,0}$. Hence the protocol is free from unspecified message receptions.

In all states the assertion $size\_of(SR) + size\_of(RS) \leq 1$ is true, therefore the channel sizes are bounded by 1.

5. Conclusion

We have presented a detailed technique for proving properties of communicating processes. It integrates two approaches: the finite state machine model and the programming language model. The technique can be used without variables and assertions, in which case it reduces to the former model. One can instead eliminate states and obtain the latter. However,

---

$$
\begin{array}{ll}
RES & : 0 < i < n\_recd : info[i] = accepted[i] \\
 & \\
INV & : s\_no = exp \Rightarrow n\_sent = n\_recd \wedge \\
 & \;\; s\_no \neq exp \Rightarrow n\_sent + 1 = n\_recd \wedge \\
 & \;\; \exists m : in\_channel_{RS}(m) \Rightarrow s\_no \neq exp \wedge \\
 & \;\; in\_channel_{SR}(i) \Rightarrow (i.seq = s\_no \wedge i.data = info[n\_sent + 1]) \wedge \\
 & \;\; s\_no,\ exp \in \{0,1\} \\
 & \\
P_{0,0} & : RES \wedge INV \wedge (size\_of(SR) + size\_of(RS) + |T| = 0) \\
P_{1,0} & : RES \wedge INV \wedge (size\_of(SR) + size\_of(RS) + |T| = 1) \\
P_{0,1} & : \textbf{false} \\
P_{1,1} & : RES \wedge INV \wedge (size\_of(SR) + size\_of(RS) + |T| = 0) \wedge \\
 & \;\; d\_item = info[n\_sent + 1] \wedge s = s\_no
\end{array}
$$

Figure 8. Assertions for the alternating bit protocol

---

the versatility of this approach arises from the fact that it allows these two models to be combined, making use of explicit states, as well as including variables and assertions. Making states explicit reduces the complexity of proofs in the programming language model; including variables and assertions eliminates the state explosion of the finite state machine model.

We have demonstrated how to model various properties of communication channels and timeout behavior, including premature and non-premature timeouts. We have also shown how to prove safety properties like freedom from deadlock and unspecified message receptions, and boundedness of channel size. The modeling technique is independent of the proof logic system used, and the axioms for the send and receive statements can be adapted to suit any other system, such as temporal logic [Pnu77, Hai82, Man82, Ngu84], to prove liveness properties like progress and termination.

This approach is illustrated by first considering a sliding window flow control protocol. With the finite state approach, proofs of the sliding window protocol have been intractable for large window sizes because of the exponential growth of the global reachability tree. We have illustrated the versatility of our model of timeouts in the proof of an alternating bit protocol that is correct only if timeouts are non-premature.

In conjunction with a theorem prover, this model is well suited to form the basis of an automated protocol verifier. Given the description of the processes participating in a protocol, the global state transition graph can be constructed automatically [Zaf80]. A theorem prover can then be used to validate the system state assertions, and to prove that the assertions imply the desired properties.

6. Acknowledgement

The authors would like to thank Abha Moitra, who read a draft of this paper, and offered many helpful suggestions. We also would like to acknowledge the comments of Mohamed Gouda and Simon Lam.

Appendix A: Proof of Sliding Window Protocol

We must show that $\{ P_{0,0} \}\ TS1\ \{ P_{0,0} \}$ is a theorem. Since $TS1$ is a send transition, the send rule applies. Let S represent the program segment

$$s\_high = s\_high \oplus 1;$$

$$high = high + 1;$$

To apply the send rule, we must find an assertion Q, such that

$$\{P_{0,0} \wedge s\_high \ominus s\_low < W\_SIZE\} \; S \; \{Q\} \qquad \textbf{(B.1)}$$

is a theorem, and

$$\{Q\} \; \textbf{send}(\; SR,\; type\colon t,\; seq\colon s\_high,\; data\colon info[\; high \;])\; \{P_{0,0}\} \qquad (B.2)$$

follows from the send axiom. Since $W\_SIZE \le M-1$, (B.1) can be proved by showing that

$$\{P_{0,0} \wedge s\_high \ominus s\_low < M-1\} \; S \; \{Q\} \qquad (B.3)$$

is a theorem. Using a proof logic system, such as in [Owi76], we prove the following theorems

$$\{RES\} \; S \; \{RES\} \qquad (i)$$

(The program segment S does not reference any variable in RES.)

$$\{P_0 \wedge s\_high \ominus s\_low < M-1\} S \{P_0\} \qquad (ii)$$

(Because s_high and $high$ are incremented in S, the assertion $s\_high = high \; mod \; M$ remains true. Because $s\_high \ominus s\_low < M - 1$, incrementing s_high increases the value of $s\_high \ominus s\_low$ by 1. Hence, $high = low + (s\_high \ominus s\_low)$ remains true.)

$$\{P_1 \wedge s\_high \ominus s\_low < M-1\}$$

$$S$$

$$\{P_1 \wedge \forall i : [in\_channel_{SR}(i) \vee in\_channel_{RS}(i)] : i.seq \neq s\_high\} \qquad (iii)$$

(Because $s\_high \ominus s\_low < M - 1$, incrementing s_high increases the size of the sender window. Since the sequence numbers of all messages in the channels were in the original sender window, they are in the new sender window. Hence, P_1 remains true. Furthermore, since s_high was not in the original sender window, none of the messages in the channels has sequence number s_high.)

$$\{P_2\} \; S \; \{P_2\} \qquad (iv)$$

$$\{P_3\} \; S \; \{P_3\} \qquad (v)$$

(S does not reference any variable in P_2 and P_3.)

$$\{P_4 \wedge s\_high \ominus s\_low < M-1\} \; S \; \{P_4 \wedge acknowledged[s\_high] = F\} \qquad (vi)$$

(Because $s\_high \ominus s\_low < M - 1$, incrementing s_high increases the size of the sender window. If $in\_window(s) = F$ after S, $in\_window(s)$ was equal to F before S. So $acknowledged[s]$ was equal to F before S. Since S does not mention *acknowledged*, its value is unchanged. Hence P_4 is true after S. Also, *s_high* was not in the original window, hence *acknowledged*[*s_high*] was equal to F. This remains true after S.)

$$\{s\_high \ominus s\_low = size\_of(SR) + size\_of(RS)\}$$
$$S$$
$$\{s\_high \ominus s\_low = size\_of(SR) + size\_of(RS) + 1\} \tag{vii}$$

From (i) through (vii) it follows that

$$\{P_{0,0} \wedge s\_high \ominus s\_low < M-1\}$$
$$S$$
$$\{INV \wedge (s\_high \ominus s\_low = size\_of(SR) + size\_of(RS) + 1) \wedge$$
$$(acknowledged[s\_high] = F) \wedge$$
$$(\forall i : [in\_channel_{SR}(i) \vee in\_channel_{RS}(i)] : i.seq \neq s\_high)\}$$

is a theorem. The post-condition above implies

$$[\, P_{0,0} \,]^{Sent_{SR}}_{Sent_{SR} \bullet \{type:MSG,\ seq:s\_high,\ data:info[high]\}} \tag{B.4}$$

Using the assertion (B.4) in the place of Q in (B.3) and (B.2) above completes the proof.

Appendix B: Detailed Proof of Transition TS1

We now show that the protocol in Figure 2 is free from deadlock, and that the accepted sequence is an initial subsequence of the data sequence, and that the number of messages in any one channel is always less than M.

From the state transition diagram in Figure 3, we see that states $S_{0,1}$, $S_{1,0}$, and $S_{1,1}$ all have outgoing send transitions that are always enabled. State $S_{0,0}$ is the only state in which a deadlock could possibly occur. The only send transition from $S_{0,0}$ is $TS1$, which has the guard $G_0 = s\_high \ominus s\_low < SW\_SIZE$. The receive transitions have guards **true**. Because the channels are FIFO, the sizes of the sets N_{SR} and N_{RS} are at most 1. At any time, the set N_{SR} has one of two values, $\varnothing$ or {MSG}. The set N_{RS} takes on values $\varnothing$, {ACK}, and {NAK}. To prove freedom from deadlock, we must show (3.2b), which here is

Notation

Let $in\_channel_c(i)$ stand for the assertion "i is the index of a message in channel c that has not been received", i.e. length($Received_c$) $<$ i $\leq$ length($Sent_c$).

Let $in\_window(s)$ stand for "the sequence number s is in the sender window", i.e. $s \in$ (s_low ... s_high].

Let $size\_of(c)$ be equal to length($Sent_c$) - length($Received_c$)

Assertions

RES: /* Result */
$\{\text{accepted[k]} \neq \varnothing \Rightarrow \text{accepted[k]} = \text{info[k]}\} \wedge \{\forall k : 0 < k \leq \text{filled} : \text{accepted[filled]} \neq \varnothing\}$

P_0 : /* Relationships between variables */
$\text{s\_low} = \text{low mod M} \wedge \text{s\_high} = \text{high mod M} \wedge \text{r\_low} = \text{filled mod M} \wedge$
$\text{high} = \text{low} + (\text{s\_high} \ominus \text{s\_low}) \wedge \text{low} \leq \text{filled} < \text{low} + \text{M}$

P_1 : /* The sequence numbers carried in data messages, acknowledgements, and negative */
/* acknowledgements all belong to the sender window. */
$\forall i : \{in\_channel_{SR}(i) \vee in\_channel_{RS}(i)\} : in\_window(i.\text{seq})$

P_2 : /* The data field of the data messages are taken from *info* in order */
$\forall i : in\_channel_{SR}(i) : i.\text{data} = \text{info[s\_low} + (i.\text{seq} \ominus \text{s\_low})]$

P_3 : /* A particular sequence number occurs at most once among all the data messages, */
/* acknowledgements, and negative acknowledgements that are still in a channel */
$\forall i,j : \{i \neq j \wedge in\_channel_{SR}(i) \wedge in\_channel_{SR}(j)\} : i.\text{seq} \neq j.\text{seq} \wedge$
$\forall i,j : \{i \neq j \wedge in\_channel_{RS}(i) \wedge in\_channel_{RS}(j)\} : i.\text{seq} \neq j.\text{seq} \wedge$
$\forall i,j : \{in\_channel_{SR}(i) \wedge in\_channel_{RS}(j)\} : i.\text{seq} \neq j.\text{seq}$

P_4 : /* If sequence number s is not on the sender window, acknowledged[s] is false, and if */
/* acknowledged[s] is true, no message on either channel carries the sequence number s */
$\{in\_window(s) = F\} \Rightarrow \{\text{acknowledged}[s] = F\} \wedge$
$\text{acknowledged}[s] \Rightarrow \forall i : \{in\_channel_{SR}(i) \vee in\_channel_{RS}(i)\} : i.\text{seq} \neq s$

$INV : RES \wedge P_0 \wedge P_1 \wedge P_2 \wedge P_3 \wedge P_4$

A_0 : $in\_window\ (\text{s\_no}) \wedge \text{d\_item} = \text{info[low} + (\text{s\_no} \ominus \text{s\_low})] \wedge \text{acknowledged[s\_no]} = F \wedge$
$\forall i : \{in\_channel_{SR}(i) \vee in\_channel_{RS}(i)\} : i.\text{seq} \neq \text{s\_no}$

A_1 : $\text{acknowledged[s\_last]} = F \wedge \forall i : \{in\_channel_{SR}(i) \vee in\_channel_{RS}(i)\} : i.\text{seq} \neq \text{s\_last}$

The assertions associated with the system states are then,

$P_{0,0} = INV \wedge \text{s\_high} \ominus \text{s\_low} = \text{size\_of}(SR) + \text{size\_of}(RS)$
$P_{0,1} = INV \wedge A_0 \wedge \text{s\_high} \ominus \text{s\_low} = \text{size\_of}(SR) + \text{size\_of}(RS) + 1$
$P_{1,0} = INV \wedge A_1 \wedge \text{s\_high} \ominus \text{s\_low} = \text{size\_of}(SR) + \text{size\_of}(RS) + 1$
$P_{1,1} = INV \wedge A_0 \wedge A_1 \wedge \text{s\_high} \ominus \text{s\_low} = \text{size\_of}(SR) + \text{size\_of}(RS) + 2$

Figure A1. Assertions for System States

$$INV \wedge s\_high \ominus s\_low = size\_of(SR) + size\_of(RS) \Rightarrow \qquad (A.1)$$

$$[s\_high \ominus s\_low < SW\_SIZE \vee \exists \text{channel } c : N_c \neq \emptyset]$$

(A.1) is true provided that the maximum sender window size $SW\_SIZE > 0$. There are two cases. The first case is if $s\_high \ominus s\_low < SW\_SIZE$. (A.1) follows immediately in this case. The other case is if $s\_high \ominus s\_low \geq SW\_SIZE$. Since $SW\_SIZE > 0$, $s\_high \ominus s\_low > 0$, and size_of(SR) + size_of(RS) > 0. Hence, $\exists$channel $c : N_c \neq \emptyset$, and (A.1) is true.

The safety property that the accepted sequence is an initial subsequence of *info* follows immediately from the fact that *RES* is true in all states.

It follows from assertions $P_{i,j}$ that $size\_of(SR) + size\_of(RS) \leq s\_high \ominus s\_low$. The modulo M difference is always in the range from 0 to $M - 1$. Hence the channel sizes are bounded by M.

REFERENCES

[Apt80] K. R. Apt, N. Francez, W. P. de Roever, "A Proof System for Communicating Sequential Processes" *ACM TOPLAS* **2**, 3, July 1980, p. 382.

[Ash71] E. Ashcroft, Z. Manna, "Formalization of Properties of Parallel Programs" *Machine Intelligence* **6**, pp. 17-41, University of Edinburgh Press, 1971.

[Boc78] G. V. Bochmann, "Finite State Description of Communicating Protocols", *Comp. Networks 2*, 4/5, October 1978.

[Boc80] G. V. Bochmann, C. A. Sunshine, "Formal Methods in Communication Protocol Design", *IEEE Trans. on Comm.*, Vol. Comm-28, No. 12, April 1980.

[Boc82] G. V. Bochmann, "A Hybrid Model and the Representation of Communication Services", *in Computer Network Architecture and Protocols*, P. E. Green, Editor, Plenum Press, 1982

[Bra83] D. Brand, P. Zafiropulo, "On Communicating Finite State Machines", *Journal of the ACM*, Vol 30, NO 2, April 1983.

[Cho83] C. H. Chow, M. G. Gouda, S. S. Lam "A Discipline for Constructing Multi-Phase Communication Protocols", *Department of Computer Sciences, Univ. of Texas at Austin, TR233*, revised October 1983.

[Dij76] E. W. Dijkstra, "A Discipline of Programming", Prentice Hall, Englewood Cliffs N. J., 1976.

[Flo67] R. W. Floyd, "Assigning Meanings to Programs", *Proc. AMS Symp. Applied Mathematics*, vol. 19. American Mathematical Society, Providence, R.I., 1967, pp. 19-31.

[Gou82] M. G. Gouda, Y. T. Yu, "Protocol Validation by Maximal Progress State Exploration", *Dept. of Computer Science. University of Texas at Austin, Tech. Report 211*, July 1982.

[Hoa69] Hoare, "An Axiomatic Basis for Computer Programming", *Commun. ACM* 21, 10, October 1969.

[Hai82] B. Hailpern, "Verifying Concurrent Programs Using Temporal Logic", Springer Verlag, Berlin-Heidelberg-New York, 1982.

[Man82] Z. Manna, A. Pnueli, "Verification of Concurrent Programs: The temporal framework", *The Correctness Problem in Computer Science*, Academic Press, London, 1982.

[Ngu84] V. Nguyen, D. Gries, S. Owicki, "A Model and Temporal Proof System for Networks of Processes", *Cornell University, Technical Report 84-651*, November 1984.

[Owi76] S. Owicki, D. Gries, "An Axiomatic Proof Technique for Parallel Programs", *Acta Informatica* 6, 1976.

[Pnu77] A. Pnueli, "The temporal Logic of Programs", *Proceedings 18th Symposium on Foundations of Computer Science*, Providence, RI, November 1977.

[Rae85] T. Räuchle, S. Toueg, "Exposure to Deadlock for Communicating Processes is Hard to Detect", *Information Processing Letters 21 (1985)*, August 1985.

[Sch84] R. D. Schlichting, F. B. Schneider, "Using Message Passing for Distributed Programming: Proof Rules and Disciplines", *ACM TOPLAS, Vol. 6, No. 3, July 1984*

[Sha83] A. U. Shankar, S. S. Lam, "An HDLC Protocol Specification and its Verification Using Image Protocols", *ACM Transactions on Computer Systems*, November 1983.

[Sun79] C. A. Sunshine, "Formal techniques for protocol specification and verification", *IEEE Computer, vol. 12*, September 1979.

[Tan81] Andrew S. Tannenbaum, "Computer Networks", *Prentice Hall. Inc., Englewood Cliffs, NJ 07632.*

[Zaf80] P. Zafiropulo, C. West, H. Rudin, D. D. Cowan "Towards Analyzing and Synthesizing Protocols.", *IEEE Trans. on Comm.*, Vol. Comm 28, No. 4, April 1980.