COMPUTER SYSTEMS LABORATORY

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

SEL-78-036

SPECIFICATION AND VERIFICATION OF A NETWORK MAIL SYSTEM

Susan S. Owicki

Technical Report No. 159

November 1978

This work was partially supported by the Air Force Office of Scientific Research under Contract No. F49620-77-C-0045.

Center for Reliable Computing

SEL-78-036

SPECIFICATION AND VERIFICATION OF A NETWORK MAIL SYSTEM

Susan S. Owicki

Technical Report No. 159

November 1978

CENTER FOR RELIABLE COMPUTING
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

To appear in the lecture notes of the International Summer School on Program Construction, Munich, 1978, Springer-Verlag.

This work was partially supported by the Air Force Office of Scientific Research under Contract No. F49620-77-C-0045.

SPECIFICATION AND VERIFICATION OF A NETWORK MAIL SYSTEM

Susan S. Owicki

Technical Report No. 159

November 1978

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

ABSTRACT

Techniques for describing and verifying modular systems are illustrated using a simple network mail problem. The design is presented in a top-down style. At each level of refinement, the specifications of the higher level are verified from the specifications of lower level components.

INDEX TERMS: Verification, Concurrency, Proof of correctness, Specifications, Networks.

1. Introduction

We wish to consider the design of a mail system that will route messages among users of a computer network. The network under consideration has a ring structure (Figure 1), in which nodes are connected by one-way communication links. Mail from a user at node i to a user at another node j must be passed around the ring from i to j. The problem is to design a subsystem of processes and monitors, running at each node, to handle the forwarding task and to receive and deliver mail for local users.

We have chosen to develop the system design in a top-down fashion. At the highest level (level 1) are the functional specifications of the mail system as a whole. These specifications, which are discussed in section 2, are a precise statement of the partial correctness requirements of the system. The first refinement, described in section 3, decomposes the system into node and link components that match the network architecture. In the next refinement, described in section 4, each node component is further refined to a set of concurrent processes communicating through buffer monitors. Each level of refinement is presented by giving specifications for the new components in the style of [1]. In addition, a partial correctness proof for the system is given as it is designed. Each level is shown to be a correct implementation of the previous level's specifications; in the last step the code of the processes and monitors is verified.

The partial-correctness specifications of the mail system state that any mail delivered is delivered to the appropriate user. Of course, it is also important that messages are eventually delivered. An informal proof that the system can be made to satisfy this requirement is given in section 5.

The network mail system in this paper is primarily intended to serve as an example of modular proof methodology. Although the overall system structure is realistic, many of the problems that arise in networks are ignored. Most of these difficulties, which include real-time constraints, synchronization protocols, and error-handling, would occur in refining the link modules introduced at level 2. They are briefly discussed when the link modules are described in section 3.

2. Level 1 Specifications: System Requirements

The functional requirements of the network mail system are given by the specifications in Figures 2 and 3. At this level, the only concern is what is to be accomplished by the system, i.e. delivery of messages to the proper destination, and not how that delivery is to be achieved.

Figure 2 defines some global types that are used in the specifications at all levels. Most important are the formats of user identifiers and messages. A userId includes both a node address and a local identifier; each user has a unique userId. Messages are passed through the system in the form of a record containing the names of the sender and intended receiver, with a text that can be an arbitrary character string.

Figure 3 gives the system's external specifications in the format that will be used for all modules: variable declarations, initial and invariant assertions, and procedure specifications. At level 1 there are two auxiliary arrays, H and C, which record the history and current state of the system. (Auxiliary variables are used in the specifications and proof, but are not actually implemented). H records the history of messages passed between modules: H[M1,M2,u,v] denotes the sequence of messages passed from M1 to M2 that have source user u and destination user v. At level 1, the only modules are the user processes (identified by userId) and the network mail system itself (NMS), but the array H will be used with other modules at lower levels. The array C is used to denote the current contents of each module: C[M,u,v] is the sequence of messages currently in M that have source u and destination v. Initially, all sequences are empty. The system invariant states that all messages sent by u addressed to v (i.e. those in H[NMS,v,u,v]) have either been delivered to v (i.e. are in H[NMS,v,u,v]) or are still in the system (i.e. are in C[NMS,u,v]). Moreover, the order in which messages are sent is preserved by the system.

The specifications for procedures send and receive indicate that they may only be called by user processes (in procedure specifications, # denotes the name of the module invoking the procedure). The effect of send is to append a message to the appropriate history. (Here H' denotes the value of H at procedure entry, and it is assumed that all elements of H not explicitly mentioned are not modified by the procedure.) The effect of receive depends on whether any mail is available for the caller. If there is, the flag valid is set to true, and a message is returned and appended to the appropriate history. Otherwise, valid is set to false, and the history is not modified.

The procedure send must also increase the sequence C[NMS,u,v] (the '(contents" of the mail system), and receive must likewise shorten C[NMS,u,v]. The effect on C is not part of the procedure entry/exit conditions, because it is not visible to the module invoking the procedures. However, it can be inferred from the entry/exit conditions and the module invariant.

These specifications illustrate a difference in notation between this paper and [1]. Rather than declaring some variables to be private to a particular module, we will use the idea of safe variables in a more informal style. A variable is safe for a module if it can only be modified by that module. The specifications and proof of a module must involve only variables that are safe for that module. Of the NMS variables, those that are safe for M1 are H[M1,M2,u,v], H[M2,M1,u,v], and C[M1,u,v] (for any M2,u,v). The values of these variables can only be changed by an action of M1, although the form of that action depends on the relationship between M1 and M2. For example, the value of H[M1,M2,u,v] could be modified by M1 calling M2.send, or by M2 calling M1.receive. Likewise, the sequence C[M1,u,v] could be extended by M2 calling M1.send or M1 calling M2.receive; and it could be shortened by M1 calling M2.send or M2 calling M1.receive.

In all cases, module specifications must use variables safely, as described in [1]. This means that free variables in the specifications of module M must obey the following rules:

1) The initial and invariant assertions may refer to any safe variable of M, e.g. C[M,u,v], H[M,M',u,v] and H[M',M,u,v] (for any M',u,v).
2) Procedure entry and exit assertions may refer to variables that are safe for the calling module, i.e. H[M,#,u,v], H[#,M,u,v] and C[#,u,v] (for any u,v).

Note that the specifications in Figure 2 obey these rules. Later refinements will use H and C in much the same way.

The functional requirements in this section are unrealistic in one major aspect: they do not require any action to be taken if mail is sent to an invalid userId. A reasonable requirement would be to return an error message to a user who sent a message with an invalid address.

A specification along these lines might have the invariant

H[u,NMS,u,v] = H[NMS,v,u,v] @ C[NMS,u,v] for valid v,

H[u,NMS,u,v] = HE[NMS,u,u,v] @ CE[NMS,u,v] @ C[NMS,u,v] for invalid v, where HE records the history of error messages between modules, and CE denotes the error messages contained in a module. The second clause of the invariant states that, for each erroneous message sent, either an error message has been received, or an error message is on its way, or the original message is still in the system. Such a specification could be implemented by having the error message initiated at v.node and returned to u using the normal message delivery system. However, we will not pursue this extension of the original specifications.

3. Level 2 Specifications: Network Architecture

3.1 Specifications

The first decomposition of the mail system fits the program to the network architecture. At each node i there is a sybsystem S[i] , and the communication line leaving node i is represented by a module L[i] . The specifications for these two component types are given in Figures 5 and 6.

First, consider the link specifications in Figure 5. The specifications are expressed in terms of the global variable H[M,L[i],u,v] and H[L[i],M,u,v]. As discussed in Section 2, these elements of the array of histories H are safe to use in the specifications of L[i] because they can only be modified as a result of actions of L[i] . The declaration of variables and the initial assertion are omitted here because no new variables are needed in the specifications.

The invariant for link L[i] states that all messages sent into the link from S[i] have been sent out to S[i⊕1] . (We will use i⊕1 and i⊖1 as abbreviations for (i+1) mod (N+1) and (i-1) mod (N+1) .) There is no buffer capacity in the link, so send and receive operations must be synchronized. The entry and exit assertions for link procedures indicate that the history sequences in H are updated appropriately, much as in the send and receive procedures of the MMS system in Figure 3. In addition, L[i].send(m) removes message m from the contents of the calling module (C[#,u,v]), and L[i].receive(m) adds m to the contents of the calling module. It way not necessary to modify C[#,u,v] in the NMS procedures send and receive because the "contents" of user processes are irrelevant to the mail system.

No further refinements of the link module are given in this paper; but in a real system, the link itself might be a complex subsystem. The link hides the details of communication devices from the rest of the system. This could involve splitting and re-assembling messages to fit a fixed-length format, synchronizing read and write operations, and recovering from transmission errors. Regardless of the complexity of the link implementations, however, the subsystem running at each node may regard the link send and receive operations as no more complex than appending and removing values in a buffer.

Figure 6 gives the specifications of the sybsystem S[i] that runs at node i. Messages arrive at S[i] from local users and from the input link L[i⊖1]. Those addressed to local users are delivered directly; the others are sent to the output link L[i]. The invariant for S[i] states that input messages (those in H[from(u),S[i],u,v]) have either been sent to the appropriate destination (i.e. are in H[S[i],to(v),u,v]) or are still in the sybsystem (i.e. in C[S[i],u,v]). The form of the invariant is quite similar to the invariant for the entire system (Figure 3); the difference is that S[i] interacts with both user processes and the links L[i⊖l] and L[i]. The procedures S[i].send and S[i].receive directly implement the corresponding level 1 procedures, with each user calling the procedures provided at his node. This is indicated by the procedures' entry assertions.

3.2 Verification

tiaving given specifications for levels 1 and 2 of the mail system, we should show that they are consistent; i.e., that the link and node modules are a valid implementation of the mail system requirements. Consistency of specifications at two levels is verified by defining the variables and procedures of the higher level in terms of the lower, and then proving that the lower level specifications imply the higher. These requirements are stated in the following definition:

Definition 1: Suppose module V is to be implemented by modules W_1, $W_2, \ldots W_k$. Let the variables of V be $\bar{v}$, the variables of $W_1, \ldots, W_k$ be $\bar{w}$ and the relationship between them be $\bar{v} = f(w)$. Then $W_1, \ldots, W_k$ correctly implement V if the following consistency conditions are satisfied.

i. $(\bigwedge_i W_i.\text{invariant}) = V.\text{Invariant}_V^{f(\bar{w})}$

ii. For each procedure p in $W_1,\ldots,W_k$ that implements a procedure q in V

a. $q.\text{entry}_V^{f(\bar{w})} \supset p.\text{entry}$

b. $p.\text{exit} \supset q.\text{exit}_V^{f(\bar{w})}$

(In the mail system, all variables in the specifications are initialized as empty sequences, so we have omitted the <u>initial</u> and <u>requires</u> clauses, described in [1], from module specifications. In the general case, these clauses would also have to be considered in proving that a lower level implementation is correct.)

Theorem 1: The level 2 specifications of modules S[i] and L[i], for i = 0,..,N (Figures 5 and 6) correctly implement the level 1 system requirements (Figure 3).

Proof: The correspondance between the names of variables and procedures of the two levels is given in Figure 7. The history of messages sent between a user u and the mail system NMS is implemented by the history of messages between u and S[u.node]. The sequence of messages in NMS from user u to user v is implemented at level 2 by the concatenation of the contents of sybsystems at v.node , v.node⊖1, ..., u.node. This reflects the fact that a message sent from u and not yet delivered to v must be at one of the nodes on the path from U to V. Finally, the send and receive procedures of level 1 are implemented at each node in level 2.

Verifying the consistency criteria for procedure entry and exit conditions is straightforward; after the substitution of variable names, the level 1 assertions are equivalent to the level 2 assertions.

Verifying the consistency of the invariants requires us to prove

$$(*)\quad (\bigwedge_i (S[i].\text{invariant} \wedge L[i].\text{invariant}) \supset \forall u,v\ (H[u,S[u.node],u,v] = H[S[v.node],v,u,v]\ @\ C[v.node,u,v]\ @\ \ldots\ @\ C[u.node,u,v])$$

Let i = u.node, j = v.node, and consider two cases for i and j. If i=j, (*) follows from

$$S[i].\text{invariant} \supset (H[u,S[i],u,v] = H[S[i],v,u,v]\ @\ C[S[i],u,v])$$

For i ≠ j, assume the left-hand-side of the implication (*). From S[i].invariant we have

H[u,S[i],u,v] = H[S[i],L[i],u,v] @ C[S[i],u,v].

Applying L[i].invariant gives

H[u,S[i],u,v] = H[L[i],S[i⊕1],u,v] @ C[S[i],u,v].

We can repeatedly apply S[k].invariant and L[k].invariant for K = i⊕1, . . .,j⊕1 to derive

H[u,S[i],u,v] = H[L[j⊕1],S[j],u,v] @ C[S[j⊕1],u,v] @...@ C[S[i],u,v]

Finally, from S[j].invariant we can derive

H[u,S[i],u,v] = H[v,S[j],u,v] @ C[S[j],u,v] @...@ C[S[i],u,v]

This completes the proof of (*) and of Theorem 1.

4. Level 3 Specifications: The Node Subsystems

4.1 Specifications

The last refinement to be presented is the decomposition of the node subsystems into processes and monitors. Figure 8 illustrates the components at each node and the flow of messages among them. There are three concurrent processes at each node, corresponding to three asynchronous activities. They are the reader process R and writer process W, which manage link communications, and a switch process SW , which routes messages to a local destination or to the output link. The processes are connected by three buffers, Swbuf , Ubuf , and Wbuf , implemented by monitors.

Specifications for level 3 components are given in Figures 9 - 14. First, consider the reader process R (Figure 9). Its invariant states that messages received from link L[i⊕1] are passed to the switch buffer Swbuf[i] . There arc no procedure specifications for a process. The specifications for the other processes (Figures 10 and 11) are similar. Process Sw[i] takes mess'ages from Swbuf[i] , sending those addressed to local users to Ubuf[i] and others to Wbuf[i] . Finally, process W[i] takes messages from Wbuf[i] and sends them to the next node via L[i] .

Specifications of the three buffers are given in Figures 12 - 14. Swbuf[i] (Figure 12) and Wbuf[i] (Figure 13) are bounded buffers of the type described in [1]. Swbuf has two "send" procedures: sendnew, called by user processes to initiate mail delivery, and send, called by the reader process to deposit messages from the input link. For both buffers, the invariant has the usual clause relating histories of messages in and out of the module, and a clause reflecting the bound on the buffer's size. In addition, the variable C[Swbuf[i],u,v] contains the subsequence of messages in Swbuf[i].buf that are addressed from u to v. (C[Wbuf[i],u,v] and Wbuf[i].buf have the same relationship.) The last clause states that the buffer only contains messages between users u and v if it is on the path from u to v. For Swbuf[i], this means that i is in the sequence u.node, u.node⊕1, ... v.node, abbreviated

i in [u.node,v.node].

For Wbuf[i], i must be in u.node, u.node⊕1, ... v.node⊕1, abbreviated

i <u>in</u> [u.node,v.node).

These limits on the buffer contents are enforced by the entry condition of send and reflected in the exit condition of receive.

The last buffer, Ubuf[i], is treated as an array of unbounded buffers, one for each local user. Presumably, these buffers are implemented using backing store which can be considered unbounded. In other respects, the specifications resemble those already considered.

4.2 Verifying Level 3 Consistency

Our next task is to verify that the level 3 specifications correctly implement the level 2 specifications of a node subsystem.

Theorem 2: The level 3 modules specified in Figures 9 - 14 are a correct implementation of the subsystem S[i] described in Figure 6.

Proof: We must show that the requirements of definition 1 are met. The correspondance between variable and procedure names from the two levels is given in Figure 15. It is easy to see that the procedure specifications are consistent, since the entry and exit conditions are identical for both levels. To show that the invariants are consistent, we must show that the conjunction of invariants for level 3 modules implies the subsystem invariant for S[i]. The reasoning involves separate consideration of four cases for u and v:

a. u.node = v.node = i

b. u.node = i $\wedge$ v.node $\neq$ i

c. u.node $\neq$ i $\wedge$ v.node = i

d. u.node $\neq$ i $\wedge$ v.node $\neq$ i

Since the four cases are treated in much the same way, we give only the proof of case a.

For u.node = v.node = i, the level 2 invariant becomes, after variable substitution,

```
(*) H[u,Swbuf[i],u,v] = H[Ubuf[i],v,u,v] @ C[Ubuf[i],u,v]
                        @ C[Sw[i],u,v] @ C[Swbuf[i],u,v]
```

Now Swbuf[i].invariant implies

```
    H[u,Swbuf[i],u,v] = H[Swbuf[i],Sw[i],u,v] @ C[Swbuf[i],u,v]
```

Applying Sw[i].invariant to expand the first term on the right-hand-side gives

```
    H[u,Swbuf[i],u,v] = H[Sw[i],Ubuf[i],u,v] @ C[Sw[i],u,v]
                        @ C[Swbuf[i],u,v]
```

Finally, applying Ubuf[i].invariant to expand the first term on the right-hand-side gives (*).

The other three cases can be proved in the same way, for example, in case d above, the level 2 invariant, after variable substitution, is

```
    H[L[i⊖1], R[i],u,v] = H[W[i],L[i],u,v] @ C[W[i],u,v]
                          @ C[Wbuf[i],u,v] @ C[Sw[i],u,v]
                          @ C[Swbuf[i],u,v] @ C[R[i],u,v].
```

This is implied by the invariants of R[i] , Swbuf[i] , Sw[i] , Wbuf[i] , and W[i] .

4.3 Verifying the Level 3 Implementation

Figures 16 - 21 contain proof outlines for the code implementing the processes and monitors of level 3. The process proofs make use of two predicates, empty and contents , defined below.

```
    empty(M: module) ≡ ∀u,v: userId (C[M,u,v] = <> )
    contents(M: module; m:message)
       ≡ ∀u,v: userId ( C[M,u,v] = if (u=m.source) and (v=m.dest)
                                     then <m>
                                     else <>                    )
```

These predicates describe the two possible states of these processes, which can contain at most one message.

For the most part, the verification of the processes and monitors is straightforward, although tedious, and is not presented here. One interesting point is that the entry conditions of Swbuf[i].send(m) requires i in (m.sourcc.node,m.dest.node]. In order to show that this entry condition is met for the procedure call in R[i] , we need to know that the message obtained from L[i01] was in the correct range. The original link specifications did not guarantee this; however, in this system the link is used in such a way that it must be true. This can be expressed by deriving specialized specifications for L[i] based on its use in the mail system. In this new specification, given in Figure 22, a stronger entry condition on L[i].send justified a stronger invariant and exit condition for L[i].receive . A formal derivation of the specialized specifications From the original ones can be obtained using techniques described in [1].

At this point we have developed a partial implementation of the mail system (without the link modules) and verified that the implementation meets the system's functional requirements. As a final step, let us consider strengthening the system requirements to imply that messages are eventually delivered.

5. Guaranteed Message Delivery

The mail system specifications given in Figure 2 require only partial correctness; they imply that if a message is received at all, it is received by the correct user. In this section we consiser two further requirements: that deadlock of the system is impossible, and that all messages are eventually delivered. (The second condition implies the first.) A set of sufficient conditions for preventing deadlock are defined and verified, and implementation methods that meet the criteria are outlined. The proofs are quite informal.

First let us consider the requirement that deadlock (a state in which all processes are blocked) cannot occur in the message system. Theorem 3 below states that deadlock is impossible if the number of undelivered messages in the system is kept smaller than its total buffer capacity. There are a number of ways of implementing the mail system to ensure that this condition is always satisfied. One approach is to delay initial processing of a message until it is certain that the network as a whole has enough buffer space to handle one more message. Several strategies have been proposed for determining, from inspection of local data at the node, when a new message can safely be allowed to enter the system (see, for example [2]). Another approach is to provide enough buffer space to hold as much mail as users can generate. In some systems, there are constraints on user behavior that keep this number small. In general, however, the number of outstanding messages may be quite large, requiring that buffers be implemented on backing store. A third approach - discarding messages when the buffer capacity is exceeded - is acceptable in some applications, but it is not consistent with our specifications.

The following theorem shows that deadlock can be avoided using any strategy that prevents the number of undelivered messages from filling all buffers to capacity.

Theorem 3. Suppose the, network mail system is implemented in such a way that the number of undelivered messages (those in C[NMS,u,v], but not in any C[Ubuf[i],u,v]) is less than $\sum_i$(Swbuf[i].bufsize + Wbuf[i].bufsize). Then whenever there is undelfvered mail in the system, at least one process is not blocked.

Proof: A process can only be blocked at monitor entry (because another process is holding the monitor) or at a monitor wait operation. The first condition can only arise when a process is executing in the monitor, so in this case at least one process is not blocked. So if all processes are blocked, they must all be blocked at wait operations. In the mail system, there are four places where this can occur:

1. At M.send, when length(M.buf) = M.bufsize, for M = Swbuf[i] or Wbuf[i].
2. At M.receive, when length(M.buf) = 0, for M = Swbuf[i] or Wbuf[i].

3. At L[i].send, when no process is executing L[i].receive.

4. At L[i].receive, when no process is executing L[i].send

The processes in the mail system form a cycle, as illustrated in figure 23. Here the processes are labelled $p_0, p_1, \ldots, p_{3N-1}$, and the monitors (excluding Ubuf) are labelled $b_0, b_1, \ldots, b_{3N-1}$. Each p_i consumes messages from b_i and produces messages for $b_{i\oplus 1}$. If deadlock occurs, each process p_i is blocked at a send to $b_{i\oplus 1}$ or a receive from b_i. Now, whether b_i is a buffer or a link, it is not possible to have both $p_{i\ominus 1}$ blocked at b_i.send and p_i blocked at b_i.receive. Since the processes from a cycle, this implies that either all processes are blocked at receive or all are blocked at send. If all are at send, then all buffers are full, and this violates the hypothesis of the theorem. So if deadlock occurs, all processes are blocked at receive. But this can only happen when all buffers are empty, and there are no undelivered messages. This completes the proof.

Even if deadlock is impossible, message delivery may not be guaranteed. For example, if deadlock is avoided by a mechanism that delays message acceptance, then some messages may be passed over repeatedly while the system delivers other messages. To preclude this possibility, the scheduling of processes and monitors must be done fairly.

Definition: A system has _fair process scheduling_ if each process makes progress at a non-zero rate unless it is blocked.

Fair scheduling for processes is natural if each process executes on its own processor. If the processes are multiprogrammed on a single processor, it is up to the multiprogramming system to ensure fair scheduling.

Definition: _A buffer implementation is fair_ if its send operations are guaranteed to terminate unless the buffer remains full forever, and its receive operations are guaranteed to terminate unless the buffer remains empty forever.

To say that a buffer monitor is fair is to imply that a process attempting to send or receive will not be passed over indefinitely in favor of other processes. If processes are competing to send elements to a buffer, one of them may be delayed for a time, but as long as the buffer does not remain full, each process will eventually complete its send. In the network system, fair scheduling of send operations is necessary for Swbuf[i], which takes input from R[i] and from local users. Fair scheduling of receive operations is needed in Ubuf[i], where user processes may compete to receive messages.

Fair buffer implementations are not difficult if the underlying implementation of monitors is fair (e.g. if monitor entry and removal from condition queues is done on a first-in-first-out basis). In this case, the buffer implementations in Figures 19 - 21 are fair. If the underlying implementation is unfair, or if the buffer scheduling policy deliberately delays some processes, e.g. in order to prevent deadlock, then accomplishing a fair buffer implementation may be more difficult.

Theorem 4. Suppose that the network mail system satisfies the conditions of Theorem 3, and that buffers and process scheduling are implemented fairly. Then if user u calls the procedure send(v,t), the message <u,v,t> will eventually reach Ubuf[v.node].

Proof: Suppose not, i.e. suppose some message <u,v,t> remains undelivered. It cannot cycle in the message system, since the invariant for Wbuf[v.node] guarantees that it cannot leave node v.node via the link. Thus it must' remain forever in some buffer b_i or process p_i. This can only happen if p_i is permanently blocked at $b_{i\theta 1}$.send By fairness, this can only happen if $b_{i\theta 1}$ remains full forever, which, in turn, can only occur if $p_{i\theta 1}$ remains blocked forever at $b_{i\theta 2}$.send Repeating this argument for $p_{i\theta 2}, \ldots, p_{i\theta 1}$, we can show that all processes are blocked. Since there is undelivered mail in the system, this is impossible, by Theorem 3. Thus all messages must eventually be delivered.

We have proved that, with fair buffers and fair process scheduling, each message is eventually delivered to the appropriate Ubuf[i]. A final requirement is that a message for user v in Ubuf[v.node] will reach v if v calls Ubuf.receive a sufficient number of times. This is easily verified, provided that Ubuf[v.node] is implemented fairly.

Combining the results of this section with those of sections 2 - 4 gives a proof of total correctness: each message is eventually delivered to the correct destination, so long as the fairness and deadlock-avoidance conditions are satisfied.

6. Summary

The purpose of this paper has been to illustrate the use of modular proofs for systems programs. Although the mail system presented here does not deal with many of the difficult problems of network communication, its overall structure is realistic. Other mail systems with modular architectures are defined in [2], [3], and [4].

The modules in this system have a common pattern, which we might call the message-passing pattern. This same sort of module appears often in other types of concurrent systems. Another common pattern, the dynamically allocated resource, is described in [5]. It is my hope that we will be able to discover a small set of patterns that account for most module structures in concurrent programs, and identify convenient ways of specifying the verifying modules which fit the patterns. If this is possible, the task of verifying large systems should be considerably simplified.

Acknowledgements: I am grateful to both Edsger Dijkstra and Leslie Lamport, whose complaints and suggestions about an earlier version of this paper led to the current form of specifications for message-passing modules.

References

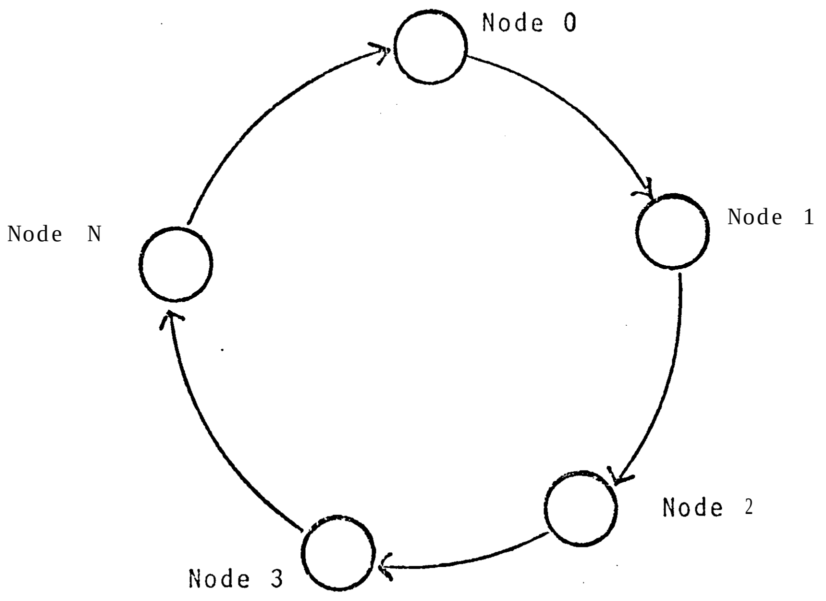
[1] Owicki, S., Specifications and Proofs for Abstract Data Types in Concurrent Programs, Lecture Notes of the International Summer School on Program Construction, Munich, 1978, Springer-Verlag (to appear). Also Stanford Computer Systems Laboratory Technical Report No. 133.

[2] Brinch Hansen, P., Network, a Multiprocessor Program. IEEE Trans on Software Engineering, v.4, no.3 (May, 1978) 194-199.

[3] Ambler, A., et al., A Language for Specification and Implementation of Verifiable Programs. Proc. of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices v. 12, n.3 (also Operating Systems Review v. 11, n.2, and Software Engineering Notes, v.2, n.2) (1977) 1-10.

[4] Andrews, G., Modula and the Design of a Message Switching-Communications System. TR78-329, Cornell University, Computer Science Dept. (1978)

[5] Owicki, S., Verifying Parallel Programs with Resource Allocation. Proc. International Conference on Math. Studies of Information Processing, Kyoto, Japan (1978)-I



'Figure 1. Ring Network Architecture

```
type nodeId = 0..N;
     localId = sequence of char;
     userId = record
                   node: nodeId;
                   uId: localId
              end;

     cstring = sequence of char;
     message = record
                   source, dest: userId;
                   text: cstring
              end;

     messageSequence = sequence of message
```

FIGURE 2. GLOBAL TYPES

```
module NMS

  var H : array [module, module, userId, userId] of messageSequence;
      c : array [module, userId, userId] of messageSequence;

  initial: H = C = <>

  invariant:  ∀u,v: userId ( H[u,NMS,u,v] = H[NMS,v,u,v] @ C[NMS,u,v] )

  procedures:

     send (u: userId; t: cstring)
        entry: #: userId
        exit:  H[#,NMS,#,u] = H'[#,NMS,#,u] @ <#,u,t>

     receive (var valid: Boolean; var u: userId; var t: cstring)
        entry: #: userId
        exit:  (valid ∧ H[NMS,#,u,#] = H'[NMS,#,u,#] @ <u,#,t>) ∨
               (~valid ∧ H[NMS,#,u,#] = H'[NMS,#,u,#])
```

Figure 3. Network Mail System (NMS) Requirements (Level 1)

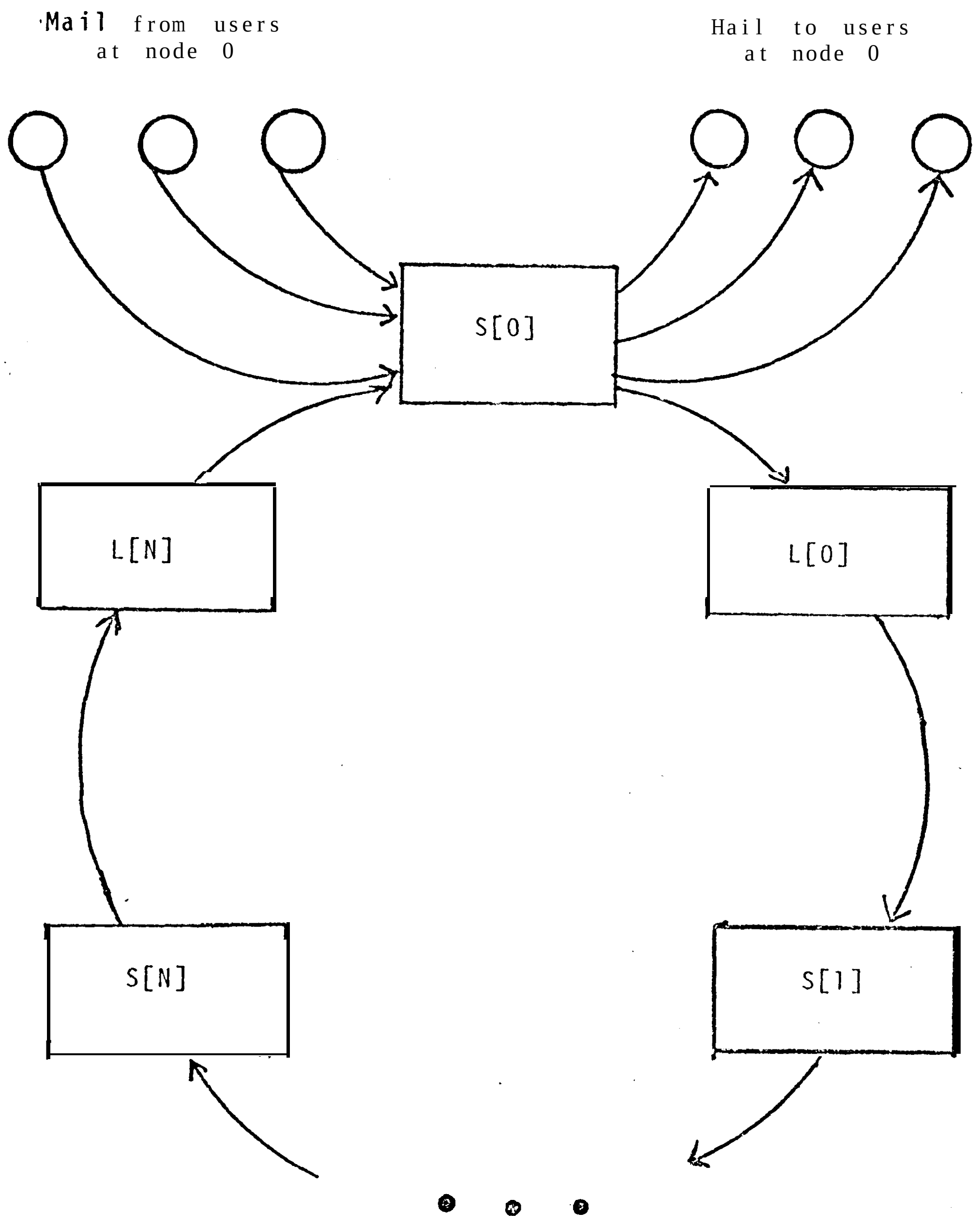


Figure 4. Level 2 Modules and Message Flow

```
module L[i]

  invariant:  Uu,v: userId ( H[S[i],L[i],u,v] = H[L[i],S[i⊕1],u,v] )

  procedures

      send:  (m: message)
         entry:  # = S[i]
         exit:   let u = m.source,
                     v = m.dest,
                 in  (H[#,L[i],u,v] = H'[#,L[i],u,v] @ <m>    ∧
                      C[#,u,v] = tail(C'[#,u,v])       )

      receive  (var m: message)
         entry:  # = S[i⊕1]
         exit:   let u = m.source,
                     v = M.dest,
                 in  (  H[L[i],#,u,v] = H'[L[i],#,u,v] @ <m>   ∧
                        C[#,u,v] = C'[#,u,v] @ <m>      )
```

Figure 5. Specifications of link module L[i]

```
module S[i]

  invariant: ∀u,v: userId
     (let  from(u) = if u.node=i then u else L[i⊖1]
              to(u) = if u.node=i then u else L[i]
      in H[from(u),S[i],u,v] = H[S[i],to(v),u,v] @ C[S[i],u,v] )

  procedures:

     send (u: userId; t: cstring)
        entry: #: userId ∧ #.node=i
        exit:  H[#,S[i],#,u] = H'[#,S[i],#,u] @ <#,u,t>

     receive (var valid: Boolean; var u: userId; var t: cstring)
        entry: #: userId ∧ #.node=i
        exit:( valid ∧ H[S[i],#,u,#] = H'[S[i],#,u,#] @ <u,#,t>)
              ∨(∼valid ∧ H[S[i],#,u,#] = H'[S[i],#,u,#] )
```

Figure 6. Specifications of Node Subsystem S[i]

In all cases u and v range over userId's

Level 1	Level 2
Variables	
H[u,NMS,u,v]	H[u,S[u.node],u,v]
H[NMS,v,u,v]	H[S[v.node],v,u,v]
C[NMS,u,v]	C[S[v.node],u,v] @ C[S[v.nodeθl,u,v] @ ... @ C[S[u.node],u,v]
Procedures	
NMS.send(u,t)	S[#.node].send(u,t)
NMS.receive(u,t)	S[#.node].receive(u,t)

Figure 7. Level 2 Implementation of Level 1 Variables and Procedures

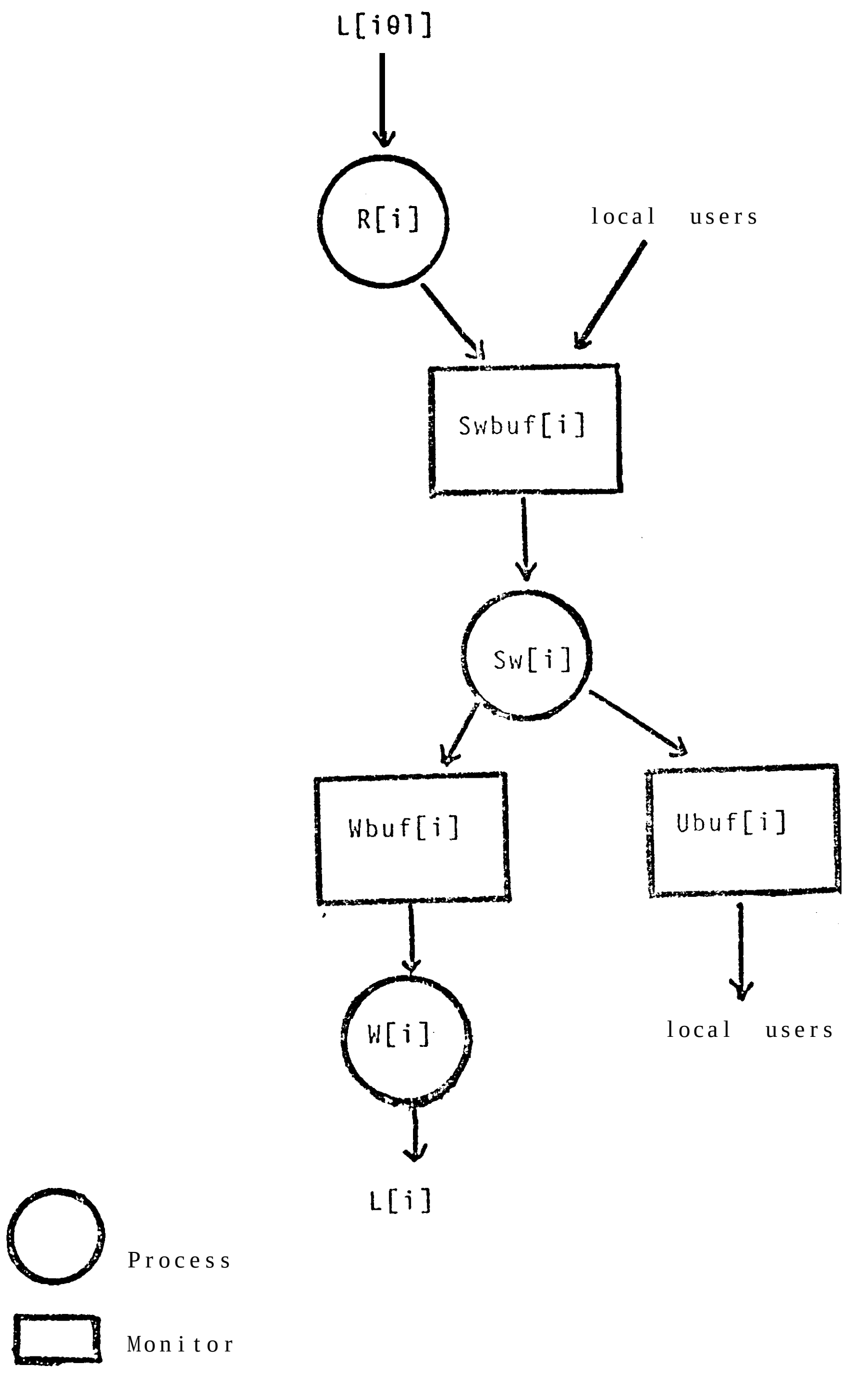


Figure 8. Level 3 Implementations of S[i]

```
process R[i]

  invariant: ∀u,v: userId
    (H[L[i⊖1],R[i],u,v] = H(R[i],Swbuf[i],u,v] @ C[R[i],u,v] )
```

Figure 9. Specifications of the Reader Process R[i]

```
process Sw[i]

  invariant: uu,v: userId
    (let to(v) = if v.node=i then v else Ubuf in
    H[Swbuf[i],Sw[i],u,v] = H[Sw[i],to[v],u,v] @ C[Sw[i],u,v] )
```

Figure 10. Specifications of the Switch Process Sw[i]

```
process W[i]

  invariant: ∀u,v: userId
    (H[Wbuf[i],W[i],u,v] = H[W[i],L[i],u,v] @ C[W[i],u,v])
```

Figure 11. Specifications of the Writer Process W[i]

```
monitor Swbuf[i]

  const bufsize

  var buf: messageSequence

  initial: buf = <>

  invariant: ∀u,v: userId
     ( let   from(u) = if u.node=i then u else R[i], in
     H[from(u),Swbuf[i],u,v] = H[Swbuf[i],Sw[i],u,v] @ C[Swbuf[i],u,v]
     ∧ length(buf) ≤ bufsize
     ∧ C[Swbuf[i],u,v] = <buf : source=u ∧ dest=v>
     ∧ ∀m: message (m in buf ⊃ i in [m.source.node,m.dest.node] ) )

  procedures

     sendnew(u: userId; t: cstring)
        entry: #: userId ∧ #.node=i
        exit: (H[#,Swbuf[i],u,v] = H'[#,Swbuf[i],u,v] @ <#,u,t>)

     send(m: message)
        entry: #=R[i] ∧ i in (m.source.node,m.dest.node]
        exit:   let u= m.source ∧ v = m.dest, in
                (H[#,Swbuf[i],u,v] = H'[#,Swbuf[i],u,v] @ <m> ∧
                ∧ C[#,u,v]= tail(C'[#,u,v] )

     receive(var m: message)
        entry: #=Sw[i]
        exit:   let u = m.source ∧ v = m.dest, in
                (H[Swbuf[i],#,u,v] = H'[Swbuf[i],#,u,v] @ <m> ∧
                ∧ C[#,u,v] = C'[#,u,v] @ <m>
                ∧ i in [u.node,v.node] )
```

Figure 12. Specifications of the Buffer Monitor Swbuf[i]

```
monitor Wbuf[i]

  const bufsize

  var buf: messageSequence

  initial: buf = <>

  invariant: ∀u,v: userId
     (H[Sw[i],Wbuf[i],u,v] = H[Wbuf[i],W[i],u,v] @ C[Wbuf[i],u,v]
      A length(buf) ≤ bufsize
      A C[Wbuf[i],u,v] = <buf : source=u A dest=v>
      A Vm: message (m in buf ⊃ i in [m.source.node,m.dest.node] ) )

  procedures

     send(m: message)
       entry: # = Sw[i]  A i in [m.source.node, m.dest.node)
       exit:  let u = m.source A v = m.dest, in
              (H[#,Wbuf[i],u,v] = H'[#,Wbuf[i],u,v] @ <m>
               A C[#,u,v] = tail(C'[#,u,v]) )

     receive(var m: message)
       entry: # = W[i]
       exit:  let u = m.source A v = m.dest, in
              (H[Wbuf[i],#,u,v] = H'[Wbuf[i],#,u,v] @ <m>
               A C[#,u,v] = C'[#,u,v] @ <m>
               A i in [u.node,v.node) )
```

Figure 13. Specifications of the Buffer Monitor Wbuf[i]

```
monitor Ubuf[i]

  var buf: array_ [localId] of messageSequence;

  initial:   buf = <>

  invariant:   uu,v: userId

      (H[Sw[i],Ubuf[i],u,v] = H[Ubuf[i],v,u,v] @ C[Ubuf[i],u,v]
      A (v.node=i ⊃ C[Ubuf[i],u,v] = <buf[v.localId] : source = u> )

  procedures

      send(m:  message)
        entry:  #=Sw[i] A m.dest.node=i
        exit:   let u = m.source A v = m.dest, in
                 (H[#,Ubuf[i],u,v] = H'[#,Ubuf[i],u,v] @ <m>
                  A C[#,u,v] = tail(C'[#,u,v]) )

      receive(var valid: Boolean; var_ u: userId; var t: cstring)
        entry:  #:userId A #.node=i
        exit:   let u = m.source A v = m.dest, in
         ( valid A H[Ubuf[i],#,u,#] = H'[Ubuf[i],#,u,#]@ <u,#,t>
         v(~valid A H[Ubuf[i],#,u,#] = H'[Ubuf[i],#,u,#] )
```

Figure 14. Specifications of the Buffer Monitor Ubuf[i]

In all cases u and v range over userId's

	Level 2	Level 3
Variables		
	H[u,S[i],u,v]	H[u,Swbuf[i],u,v]
	H[L[iθl[,S[i],u,v]	H[L[iθl],R[i],u,v]
	H[S[i],v,u,v]	H[Ubuf[i],v,u,v]
	H[S[i],L[i],u,v]	H[W[i],L[i],u,v]
	C[S[i],u,v]	Y(v) @ C[Sw[i],u,v] @ C[Swbuf[i],u,v] @ X(u)

```
Where
 X(u) = if  u.node=i
          then <>
          else C[R[i],u,v]
 Y(v) = if  v.node=i
          then C[Ubuf[i],u,v]
          else C[Wbuf[i],u,v]
                 @      C[W[i],u,v]
```

	Level 2	Level 3
Procedures		
	S[i].send(u,t)	Swbuf[i].sendnew(u,t)
	S[i].receive(val,u,t)	Ubuf[i].receive(val,u,t)

Figure 15. Level 3 Implementation of Level 2 Variables and Procedures

```
process R[i]

  var m: message;

  begin
     {invariant ∧ empty(R[i]) }
     while true do begin
        (invariant ∧ empty(R[i]) }
        L[i⊖1].receive(m) ;
        {invariant ∧ i in (m.source.node,m.dest.node) ∧
           contents(R[i],m) }
        Swbuf[i].send(m);
        {invariant ∧ empty(R[i]) }
     end
  end
```

Figure 16. Proof Outline for the Reader Process R[i]

```
process Sw[i]

  var m: message

  begin
     {invariant ∧ empty(Sw[i]) }
     while true do begin
     (invariant ∧ empty(Sw[i]) }
     Swbuf[i].receive(m);
     I-invariant ∧ i in [m.source,m.dest] ∧
      contents(Sw[i], m ) }
     if m.dest.node = v
        then Ubuf.send(m)
        else Wbuf.send(m)
     {invariant ∧ empty(Sw[i]) }
     end
  end
```

Figure 17. Proof Outline for the Switch Process Sw[i]

```
process W[i]

  var m: message;

  begin
     {invariant ∧ empty(W[i]) }
     while true do begin
        {invariant ∧ empty(W[i]) }
        wbuf[i].receive(m);
        (invariant ∧ i in [m.source.node,m.dest.node] ∧
          contents(W[i],m) }
        L[i].send(m);
        {invariant ∧ empty(W[i]) }
  end
end
```

Figure 18. Proof Outline for the Writer Process W[i]

```
monitor Swbuf[i]
  const bufsize =...
  var buf: messageSequence;
      nonempty, nonfull: condition;
  procedure sendnew(u: userId; t: cstring);
     begin
        {invariant ∧ sendnew.entry}
        if length (buf) = bufsize then nonfull.wait;
        {invariant ∧ sendnew.entry ∧ length(buf) < bcfsize }
        buf := buf @ <#,u,t>;
        H[#,Swbuf[i],#,u] : = H[#,Swbuf[i],#,u] @ <#,u,t>
        C[Swbuf[i],#,u] : = C[Swbuf[i],#,u] @ <#,u,t>
        {invariant ∧ sendnew.exit ∧ length(buf) > 0 }
        nonempty.signal;
        {invariant ∧ sendnew.exit }
     end

  procedure send(m: message);
     var u,v: userId;
     begin
        {invariant ∧ send.entry }
        if length(buf) = bufsize then nonfull.wait;
        (invariant ∧ i in (m.source.node,m.dest.node]
         ∧ #=R[i] ∧ length(buf)<bufsize }
        buf : = buf @ <#,u,t>;
        u : = m-source; v := m.dest;
        H[#,Swbuf[i],u,v] := H[#,Swbuf[i],u,v] @ <m>;
        C[#,u,v] := tail(C[#,u,v]);
        C[Swbuf[i],u,v] := C[Swbuf[i],u,v] @ <m>;
        {invariant ∧ send.exit ∧ length(buf) > 0 }
        nonempty.signal
        {invariant ∧ send.exit}
     end
```

Figure 19. Proof Outline for the Buffer Monitor Swbuf[i]
(Cont. on next page)

```
procedure receive(var m: message);
   var u,v: userId;
   begin
      {invariant ∧ receive.entry}
      if length(buf) = 0 then noncmpty.wait;
      {invariant ∧ #= Swbuf[i] ∧ length(buf) > 0 }
      m := head(buf); buf: = tail(buf);
      u : = m.source; v := m.dest;
      H[Swbuf[i],#,u,v] := H[Swbuf[i],#,u,v] @ <m>;
      C[#,u,v] := C[#,u,v] @ <m>;
      C[Swbuf[i],u,v] := tail(C[Swbuf[i],u,v]);
      {invariant ∧ receive.exit ∧ length(buf) < bufsize }
      nonfull.signal;
      {invariant ∧ receive.exit }
   end
begin
   buf : = <>
end;
```

Figure 19. Proof Outline for the Guffer Monitor Swbuf[i]

```
monitor Wbuf[i]
   const bufsizc = . .
   var buf: messageSequence;
       nonempty, nonfull: condition;
procedure send(m: message);
   var u,v: userId;
   begin
      {invariant A send.entry }
      if length(buf) = bufsize then nonfull.wait;
      {invariant A #=Sw[i] A i in [m.sourcc.node,m.dest.node)
       A length(buf) < bufsize }
      buf := buf @ <m>;
      u := m.source;   v := m.dest;
      H[#,Wbuf[i],u,v]:= H[#,Wbuf[i],u,v] @ <m>
      C[#,u,v] := tail(C[#,u,v])
      C[Wbuf[i],u,v] := C[Wbuf[i],u,v] @ <m>
      {invariant A send.exit A length(buf) > 0 }
      nonempty.signal;
. .   C-invariant A send.exit}
   end;

procedure receive(var m: message)
   var u,v: userId;
   begin
      (invariant A receive.entry }
      if length(buf) = 0 then nonempty.wait;
      {invariant A #=W[i] A length(buf) > 0 }
      m := head(buf); buf := tail(buf);
      u := m.source;   v := m.dest;
      H[Wbuf[i],#,u,v] := H[wbuf[i],#,u,v] @ <m>;
      C[#,u,v] := C[#,u,v] @ <m>
      C[Wbuf[i],u,v] := tail(Cbuf[i],u,v]);
      {invariant A receive.exit A length(buf) > bufsize}
      nonfull.signal
      {invariant A receive.exit}
   end
begin
   buf := <>
end
```

Figure 20. Proof Outline for the Buffer Monitor Wbuf[i]

```
monitor Ubuf[i]
    var buf: array [localid] of messageSequence;
    procedure send(m: message);
        var u, v: userId;
        begin
            {invariant A send.entry }
            u := m.source; v := m.dest;
            buf[v.localId] := buf[v.localId] @ <m>;
            H[#,Ubuf[i],u,v] := H[#,Ubuf[i],u,v] @ <m>;
            C[#,u,v] := tail(C[#,u,v]);
            C[Ubuf[i],u,v] := C[Ubuf[i],u,v] @ <m>;
            {invariant A send.exit }
        end;
    procedure receive (var valid: Boolean; var u: userId; var t: cstring)
        var m: message;
        begin
            (invariant A receive.entry }
            if length(buf[#.localId]) = 0
            then valid := false
            else begin ,
              m := head(buf[f.localId]);
              buf[#.localId] := tail(buf[#.localId]);
              u := m.source; t := m. text;
              valid := true;
              H[Ubuf[i],#,u,#] := H[ubuf[i],#,u,#] @ <m>;
              C[Ubuf[i],u,#] := tail(C[Ubuf[i],u,#])
          end
          (invariant A receive.exit }
      end;
  begin
      buf := <>
  end
```

Figure 21. Proof Outline for Buffer Monitor Ubuf[i]

```
module  L[i]
    invariant:   ∀u,v: userId (H[S[i],L[i],u,v] = H[L[i], S[i⊕1],u,v]
                 ∧ ( ( i not in [u.node,v.node)⊃ H[S[i],L[i],u,v] =<>))

procedures
    send(m:message)
       entry:     # = S[i] ∧ i in  [m.source.node,m.dest.node)
       exit:     let u = m.source,
                     v = m.dest,
                 in (H[#,L[i],u,v] = H'[#,L[i],u,v] @ <m> ∧
                      C[#,u,v] = tail(C'[#,u,v])

    receive(var m: message)
       entry: # = S[i⊕1]
       exit:     let u = m.source
                      v = m.dest
                 in ( i in [u.node,v.node) ∧
                      H[L[i],#,u,v] = H'[L[i],#,u,v] @ <m> ∧
                      C[#,u,v] = C'[#,u,v] @ <m> )
```

Figure 22. Adapted Specifications of L[i] (for Level 3 Verification)

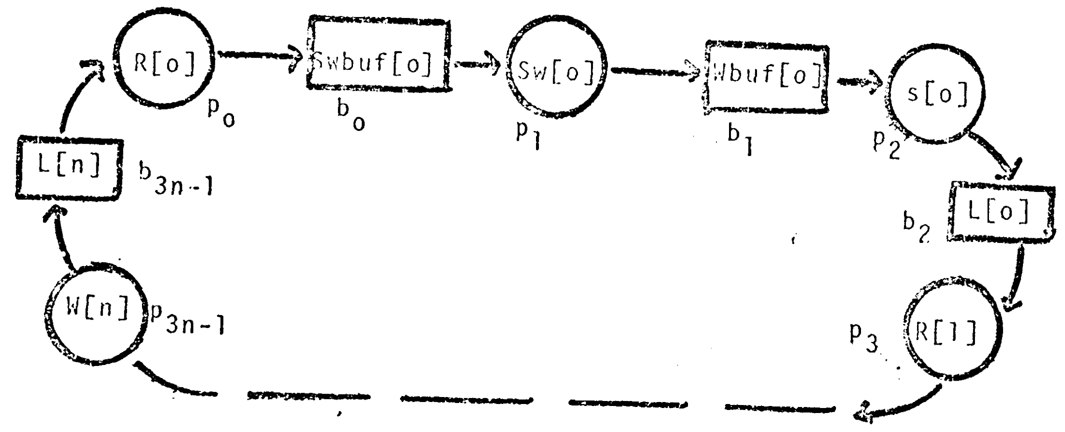


Figure 23. Mail System Processes and Monitors