A DISTRIBUTED PATH ALGORITHM AND ITS CORRECTNESS PROOF*

David D. Wright
Fred B. Schneider

TR 83-556
June 1983

Department of Computer Science
Cornell University
Ithaca, New York 14853

*This work is supported in part by NSF Grant MCS-8103605

A Distributed Path Algorithm and its Correctness Proof*

David D. Wright
Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

June 27, 1983

ABSTRACT

A distributed program is developed to allow a process in a network to determine a path from itself to any other process, assuming that the topology of the entire network is not known to any process and that each process knows only the names of the processes to which it is directly connected. The solution, written in CSP, is proved correct and deadlock-free.

*This work is supported in part by NSF Grant MCS-8103605.

1. Introduction

Consider a network of processes connected by unidirectional communication links. Such a network can be modeled by a directed graph in which processes are represented by nodes and communication links by edges. Let the graph representing the network be strongly connected, and assume each process has a unique name that is known to all other processes in the network. Further, assume that the topology of the entire network is not known to any process, although each process does know the names of the processes to which it is directly connected. Desired is a distributed program that will allow a process to determine a path—a sequence of names of connected processes—from itself to any other process.

The program will be described in a variant of Hoare's Communicating Sequential Processes notation (CSP) [Hoar78], extended to include output guards. The solution is proved correct and deadlock-free using the proof method of Levin and Gries [LeGr81].

2. The Environment

Given are processes P_1, P_2, ..., P_N. Associated with each process P_i are the *successors* of P_i, a set T_i of processes to which P_i can transmit messages, and the *predecessors* of P_i, a set R_i of processes from which P_i can receive messages. We require $j \in R_i \Leftrightarrow i \in T_j$.

A *path* is a finite sequence of process names. A sequence consisting of a_0, a_1, ..., a_n in that order will be denoted by $<a_0, a_1, ..., a_n>$, the null sequence by $<>$. In order to construct and examine sequences, two functions will be useful:

$$first(S) = \begin{cases} a_0 & \text{if } S = <a_0, ..., a_n> \\ \Phi & \text{if } S = <> \end{cases}$$

$$last(S) = \begin{cases} a_n & \text{if } S = <a_0, ..., a_n> \\ \Phi & \text{if } S = <> \end{cases}$$

where Φ is a name distinct from any P_i. In addition, the sequence obtained by concatenating an element i to the end of a sequence $<a_0, ..., a_n>$ will be denoted by $<a_0, ..., a_n> \cdot i$.

If at termination, process P_i is to have a path from S to D in a variable p_i, the desired postcondition of P_i must include the predicate $path(p_i,S,D)$, where

$$path(p,x,y) \equiv [first(p){=}x \wedge last(p){=}y \\ \wedge (p{=}{<}a_0, \ldots, a_n{>} \Rightarrow ((\forall i\colon 0{<}i{\le}n\colon a_i \in T_{i\text{-}1}) \\ \wedge (\forall i\colon 0{\le}i{<}n\colon a_i \in R_{i+1})))]$$

That is, p is a feasible path from x to y.

3. The Algorithm

We now develop a program to solve the problem described above. The proof of the program is detailed in sections 4 and 5.

Execution of the program can be viewed as being divided into two phases. The first phase commences after some process P_S has selected another process, say P_D, with which it wishes to communicate. P_S then sends information about a partial path and the desired destination (P_D) to each member of T_S. These processes, upon receipt of such a message, concatenate their names to the partial paths they receive, and send the results to their successors, etc. A wave of partial paths emanates from P_S through the network. Thus, the first phase is characterized by no process possessing a complete path.

Since the graph representing the network is strongly connected, eventually the wave of messages will reach P_D. Like any other process, P_D concatenates its name to a path. At this point, the system enters the second phase, which is characterized by at least one process having a complete path. P_D sends this complete path to all members of T_D, which in turn send it to their successors, etc. This second wave spreading out from P_D will eventually reach P_S, since the graph is strongly connected. (P_D cannot simply send the complete path back along its own reverse, since the network edges are directed.)

Once a process has sent the complete path to all of its successors, it can terminate. In CSP, as originally defined in [Hoar78], this would pose no problem, since a communication guard that names a terminated process is false. However, the proof system of [LeGr81] is for a variant of CSP that does not support this distributed termination facility. In this variant, a guard evaluates to false only when its Boolean component is false, so termination of a guarded command loop must be done based on local information only. Since a process is sending a complete path to all of its successors, this suggests that it should also wait to receive a complete path from all of its predecessors. Once this happens, the process will be involved in no further communications and can terminate without causing deadlock. The program developed in this section does exactly this.

To develop a program from the above, we first make several observations:

(1) A process P_i is involved in at most $|R_i|+2|T_i|+1$ communications. It could send a partial and a complete path to each member of T_i, receive a complete path from each member of R_i, and receive a partial path from some member of R_i. This suggests using a loop that will iterate at most $|R_i|+2|T_i|+1$ times.

(2) A process need receive at most one partial path. Indeed, allowing it to receive more than one could result in a program that does not terminate, with two processes eternally exchanging and extending partial paths.

(3) A complete path is a partial path that has the destination as its last element. Each process P_i will use two different variables: pp_i to hold a partial path and p_i to hold a complete path. Given this, P_i will be in the first phase if $pp_i \neq <>$ and in the second if $p_i \neq <>$.

For a process to terminate, it must send a complete path to all of its successors and receive one from all of its predecessors. We can use a set of Boolean variables $\{s_i^j,\ j \in T_i\}$, that have value true iff a complete path has been sent by P_i to P_j, and another set $\{r_j^i,\ j \in R_i\}$, that have value true iff a complete path has been received by P_i from P_j. The strengthened postcondition for process P_i is:

Q_i: $path(p_i,S,D) \wedge (\forall j \in T_i\colon s_i^j) \wedge (\forall j \in R_i\colon r_j^i)$

We can now write a program:

$$
\begin{array}{l}
P \;::\; \mathbf{for}\ i := 1\ \mathbf{to}\ n \\
\qquad \mathbf{for}\ j \in T_i\!:\ s_i^j := \mathit{false}\ \mathbf{rof}\,; \\
\qquad \mathbf{for}\ j \in R_i\!:\ r_j^i := \mathit{false}\ \mathbf{rof} \\
\qquad \mathbf{rof}\,; \\
\quad [\, P_1 \;||\; \cdots \;||\; P_N \,]
\end{array}
$$

where

$$
\begin{array}{l}
P_i :: \mathbf{if}\ \ i = S \ \rightarrow\ p_i,\, pp_i,\, d_i := <>,\ <S>,\ D \\
\quad [\!]\ \ i \neq S \ \rightarrow\ p_i,\, pp_i,\, d_i := <>,\ <>,\ \Phi \\
\quad \mathbf{fi}; \\
\quad \mathbf{do}\ \underset{j \in R_i}{[\!]}\ p_i = <> \wedge pp_i = <>;\ P_j?(pp_i, d_i) \rightarrow\ \mathbf{if}\ \ d_i = i \rightarrow p_i,\, pp_i := pp_i \cdot i,\ <> \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [\!]\ \ d_i \neq i \rightarrow pp_i := pp_i \cdot i \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{fi} \\
\qquad \underset{j \in T_i}{[\!]}\ pp_i \neq <>;\ P_j!(pp_i, d_i) \rightarrow\ \mathit{skip} \\
\qquad \underset{j \in R_i}{[\!]}\ \neg\, r_j^i;\ P_j?(p_i, r_j^i) \rightarrow\ pp_i := <> \\
\qquad \underset{j \in T_i}{[\!]}\ \neg\, s_i^j \wedge p_i \neq <>;\ P_j!(p_i, \mathit{true}) \rightarrow\ s_i^j := \mathit{true} \\
\quad \mathbf{od}
\end{array}
$$

4. The Weak Correctness Proof

In the logic of [LeGr81], a proof of a CSP program consists of a weak correctness proof, which shows correctness in the absence of deadlock, and a proof that the program is deadlock-free. A weak correctness proof is itself divided into a sequential proof, a satisfaction proof, and a non-interference proof.

A sequential proof is an annotation of the program using axioms and rules of inference, in which any assertion may appear as the postcondition of a communication statement. A satisfaction proof shows that such assertions are valid. A non-interference proof is used to show that assertions involving shared auxiliary variables are valid in light of concurrent execution.[1] Our proof of P_i uses a set of auxiliary variables $\{a_i^j,\ j \in T_i\}$, which are initially false. Variable a_i^j will be set to true iff a path (partial or complete) is sent from P_i to P_j. That is,

[1]Although shared program variables are not allowed in a CSP program, shared auxiliary variables are allowed.

$a_i^j \Rightarrow pp_j \neq <> \vee p_j \neq <>$

4.1. The Sequential Proof

Here is the annotated program P_i with auxiliary variables added.

P_i ::

$$\{DL1_i \wedge (\forall j \in T_i\colon \neg s_i^j) \wedge (\forall j \in R_i\colon \neg r_j^i \wedge \neg a_j^i)\}$$

$$\mathbf{if}\ \ i = S \rightarrow p_i,\ pp_i,\ d_i := <>,\ <S>,\ D$$

$$[\,]\ \ i \neq S \rightarrow p_i,\ pp_i,\ d_i := <>,\ <>,\ \Phi$$

$$\mathbf{fi};$$

$$\{INV_i \wedge DL_i\}$$

$$\mathbf{do}\ \underset{j \in R_i}{[\,]}\ p_i = <> \wedge pp_i = <>;\ P_j?(pp_i, d_i, a_j^i) \rightarrow \{INV1_i \wedge INV2_i \wedge DL_i \wedge d_i = D \wedge path(pp_i, S, j) \wedge p_i = <> \wedge a_j^i\}$$

$$\mathbf{if}\ \ d_i = i \rightarrow p_i,\ pp_i := pp_i \cdot i,\ <>$$

$$[\,]\ \ d_i \neq i \rightarrow pp_i := pp_i \cdot i$$

$$\mathbf{fi}$$

$$\{INV_i \wedge DL_i\}$$

$$\underset{j \in T_i}{[\,]}\ pp_i \neq <>;\ P_j!(pp_i, d_i, true) \rightarrow skip \quad \{INV_i \wedge DL_i\}$$

$$\underset{j \in R_i}{[\,]}\ \neg r_j^i;\ P_j?(p_i, r_j^i, a_j^i) \rightarrow \{INV2_i \wedge INV3_i \wedge DL_i \wedge path(p_i, S, D)\}$$

$$pp_i := <> \quad \{INV_i \wedge DL_i\}$$

$$\underset{j \in T_i}{[\,]}\ \neg s_i^j \wedge p_i \neq <>;\ P_j!(p_i, true, true) \rightarrow \{INV_i \wedge DL2_i \wedge DL3_i \wedge r_i^j \wedge p_i \neq <> \wedge (\forall k \in T_i\colon k \neq j\colon \neg s_i^k \Rightarrow \neg r_i^k)\}$$

$$s_i^j := true \quad \{INV_i \wedge DL_i\}$$

od

$$\{INV_i \wedge DL_i \wedge (p_i \neq <> \vee pp_i \neq <>) \wedge pp_i = <> \wedge (\forall j \in R_i\colon r_j^i) \wedge (\forall j \in T_i\colon s_i^j \vee p_i = <>)\}$$

$$\{INV_i \wedge DL_i \wedge pp_i = <> \wedge p_i \neq <> \wedge (\forall j \in R_i\colon r_j^i) \wedge (\forall j \in T_i\colon s_i^j)\}$$

$$\{path(p_i, S, D) \wedge (\forall j \in R_i\colon r_j^i) \wedge (\forall j \in T_i\colon s_i^j)\}$$

where

$$INV_i \equiv INV1_i\text{: } (p_i{=}{<>} \lor pp_i{=}{<>}) \land$$

$$INV2_i\text{: } (p_i{=}{<>} \lor path(p_i,S,D)) \land$$

$$INV3_i\text{: } (pp_i{=}{<>} \lor (path(pp_i,S,i) \land d_i{=}D))$$

$$DL_i \equiv DL1_i\text{: } (\forall j \in T_i\text{: } s_i^j = r_i^j) \land$$

$$DL2_i\text{: } (p_i{=}{<>} \Rightarrow (\forall j \in R_i\text{: } \neg r_j^i) \land (\forall j \in T_i\text{: } \neg s_i^j)) \land$$

$$DL3_i\text{: } (\forall k \in R_i\text{: } a_k^i \Rightarrow (p_i{\neq}{<>} \lor pp_i{\neq}{<>}))$$

4.2. The Satisfaction Proof

A satisfaction proof shows the validity of the postconditions of communication actions. Suppose that process A contains $B!(\overline{e})$ and process B contains $A?(\overline{x})$, where $\overline{e}$ is a vector of expressions and $\overline{x}$ is a vector of variables. Suppose also that $\overline{e}$ and $\overline{x}$ are of the same length and that their corresponding elements are of the same type. In this case, $B!(\overline{e})$ and $A?(\overline{x})$ form a *matching pair.* To show that communication establishes the desired postconditions, the following rule is used:

If $\{P\}\ B!(\overline{e})\ \{Q\}$ and $\{R\}\ A?(\overline{x})\ \{V\}$ form a matching pair, then $(P \land R) \Rightarrow (Q \land V)_{\overline{e}}^{\overline{x}}$.

The program has only two matching pairs: the send/receive of a partial path and the send/receive of a complete path. We consider each below for process P_i sending to process P_j.

Sending a partial path. The above rule requires us to show

$$(INV_i \land DL_i \land pp_i{\neq}{<>} \land INV_j \land DL_j \land p_j{=}{<>} \land pp_j{=}{<>}) \Rightarrow$$

$$(INV_i \land DL_i \land INV1_j \land INV2_j \land DL_j \land d_j{=}D \land path(pp_j,S,i) \land \ p_j{=}{<>} \land a_i^j)_{pp_i,\ d_i,\ true}^{pp_j,\ d_j,\ a_i^j}$$

From the antecedent, we can deduce

$$INV_i \land DL_i \land INV_j \land DL_j \land d_i{=}D \ \land p_i{=}{<>} \land p_j{=}{<>} \land pp_j{=}{<>} \land path(pp_i,S,i),$$

which implies the result of the textual substitution

$$INV_i \land DL_i \land (p_j{=}{<>} \lor pp_i{=}{<>}) \land INV2_j \land DL1_j \land DL2_j \land$$

$$(\forall k \in R_j\text{: } a_k^j \Rightarrow (p_j{\neq}{<>} \lor pp_i{\neq}{<>}))_{true}^{a_i^j} \land d_i{=}D \land path(pp_i,S,i) \land p_j{=}{<>} \land true.$$

Sending a complete path. The rule requires us to show

$$(INV_i \wedge DL_i \wedge \neg s_i^j \wedge p_i \neq <> \wedge INV_j \wedge DL_j \wedge \neg r_i^j) \Rightarrow$$

$$(INV_i \wedge DL2_i \wedge DL3_i \wedge r_i^j \wedge p_i \neq <> \wedge (\forall k \in T_i \colon k \neq j \colon \neg s_i^k \Rightarrow \neg r_i^k)$$

$$\wedge\ INV2_j \wedge INV3_j \wedge DL_j \wedge path(p_j, S, D))^{p_j, r_i^j, a_i^j}_{p_i, true, true}$$

From the antecedent, we can deduce

$$INV_i \wedge DL_i \wedge \neg s_i^j \wedge path(p_i, S, D) \wedge INV_j \wedge DL_j \wedge \neg r_i^j$$

which implies the result of the textual substitution

$$INV_i \wedge DL2_i \wedge DL3_i \wedge true \wedge p_i \neq <> \wedge (\forall k \in T_i \colon k \neq j \colon \neg s_i^k \Rightarrow \neg r_i^k) \wedge$$

$$INV2_i \wedge INV3_j \wedge DL1_j \wedge (DL2_j)^{p_j, a_i^j}_{p_i, true} \wedge (DL3_j)^{p_j, a_i^j}_{p_i, true} \wedge path(p_i, S, D).$$

4.3. The Non-Interference Proof

Constructing a non-interference proof can be a formidable task, since it requires showing that no assignment statement or communication action in any process can invalidate any assertion in any other process. Fortunately, this task is greatly simplified if shared variables are synchronously altered. A variable v is *synchronously altered in process* P_i if its value is only changed by assignments to v in P_i, input commands in P_i, or input commands in other processes that match output commands in P_i. Under these restrictions, no process can asynchronously change the value of such a variable. Thus, synchronously altered variables are not subject to interference.

All variables referenced in assertions in P_i are synchronously altered, so the proof of non-interference is immediate.

4.4. Proving Termination

To prove total correctness, we must show that the iterative construct in each P_i terminates. The termination function of P_i is:

$$Term_i \colon\ (\mathbf{N}\, j \in T_i \colon \neg s_i^j) + (\mathbf{N}\, j \in T_i \colon \neg a_i^j) + (\mathbf{N}\, j \in R_i \colon \neg r_j^i) + \beta_i$$

where

N denotes "number of," and

$$\beta_i \equiv \textbf{if}\ pp_i=p_i=<> \rightarrow 1\ [\!]\ pp_i\neq<> \vee p_i\neq<> \rightarrow 0\ \textbf{fi}.$$

To show that this function has the desired properties, we must prove

(1) $Term_i = 0$ implies all guards in the loop of P_i are false, and

(2) each iteration of P_i decreases the value of $Term_i$ by at least 1

For the first point, note that

$$Term_i=0 \Rightarrow (\forall j \in T_i\colon s_i^j) \wedge (\forall j \in R_i\colon r_j^i) \wedge (pp_i\neq<> \vee p_i\neq<>)$$

so all but the second guard of the iterative construct will be false. In addition, from $DL2_i$ and $Term_i = 0$, we can deduce $p_i\neq<>$, which implies $pp_i=<>$ by $INV1_i$. Hence the second guard is also false and the loop will terminate.

It is clear that sending a complete path will decrease $Term_i$ by at least 1, as the path can only be sent if $\neg s_i^j$, and s_i^j is set to true immediately thereafter. A complete path can only be received if $\neg r_j^i$, and receiving the path makes r_j^i true, so this will also decrease $Term_i$. A partial path can only be received if $p_i=<> \wedge pp_i=<>$ (i.e. $\beta_i = 1$), and receiving it will set β_i to 0. Finally, a partial path can only be sent from P_i to P_j if $pp_j=<> \wedge p_j=<>$. From $DL3_j$ we can then infer $\neg a_i^j$, and since the send will establish a_i^j, $Term_i$ will decrease by 1.

Note: No process name appears more than once in any p_i at any time. Informally, we can see this because a process concatenates its name to a path only within the first guard of the iterative construct. This will establish $(pp_i\neq<> \vee p_i\neq<>)$. Once this has been established, no later action of the program will falsify it, so the concatenation can take place only once. However, since acyclicity of p_i is not essential, we have left the observation informal. *End of Note.*

5. The Deadlock-Freedom Proof

A formal proof of strong correctness consists of a weak correctness proof and a proof that progress is possible in any possible configuration (that is, in any program state other than complete termination).

The latter proof is carried out by showing that the assumption of deadlock implies a contradiction.

In a CSP program, processes can only be blocked at communication commands, at the end of parallel constructs, and at alternative or iterative commands that contain communication guards. Unfortunately, in general the number of possible configurations is exponential in the number of processes in a program. Because of this, a formal proof is usually too large to aid in understanding a program. Hence, the proof presented here is informal.

The proof will consist of considering each possible blocking point in turn and showing that the assumption that a process can be permanently blocked there results in a contradiction. (This method is suggested in [LeGr81].)

(1) Suppose that a process P_i is permanently blocked waiting to send a complete path to some process P_j, $j \in T_i$, so $\neg s_i^j \wedge p_i \neq <>$. Since P_j is not prepared to receive, it must be that r_i^j. But from $DL1_i$, we can conclude $\neg r_i^j$, a contradiction.

(2) Suppose that $p_i = <> \wedge pp_i = <>$ and P_i is permanently blocked. Clearly, P_i is ready to receive a partial path from any member of R_i. From $DL2_i$ we can infer $(\forall j \in R_i: \neg r_j^i)$, so P_i is also ready to receive a complete path from any member of R_i. We know $(\forall j \in R_i: \neg s_j^i)$ by $DL1_j$, and since the postcondition of any P_j includes s_j^i, none of the P_j can have terminated. Thus, all the P_j are blocked in their loops and not ready to send, so $(\forall j \in R_i: p_j = <> \wedge pp_j = <>)$. That is, all the P_j are blocked in the same fashion as P_i, and by induction, all processes must be so blocked. But this is impossible, because in P_S, initially $pp_S = <S>$, and we observe that thereafter P_S maintains $(pp_S \neq <> \vee p_S \neq <>)$.

(3) Suppose $pp_i \neq <>$ and P_i is permanently blocked. By $INV1_i$ $p_i = <>$, so by $DL2_i$ $(\forall j \in R_i: \neg r_j^i)$ and by $DL1_j$, $j \in R_i$, $(\forall j \in R_i: \neg s_j^i)$. Since P_i is permanently blocked, it must be that no member of R_i will ever be prepared to send to P_i. Thus $p_j = <>$ and all the P_j are permanently blocked at the last guarded command in the loop. (They cannot have terminated by $DL2_j$, since the postcondition of P_j includes s_j^i.) It cannot be that $pp_j = <>$, as the previous proof shows that P_j would not then be permanently blocked, so $pp_j \neq <>$ and all the P_j are blocked in the same way as P_i, and by induction, all processes must be so blocked. But this is impossible, as P_D has

$pp_D \neq <>$ only immediately after receiving a partial path, and since it immediately thereafter sets pp_D to $<>$ and p_D to the complete path, P_D will never block with $pp_D \neq <>$.

(4) Suppose that $\neg r_j^i$ for some $j \in R_i$ and P_i is permanently blocked waiting to receive a complete path from P_j. As in the previous proof, P_j has not terminated, as we can deduce $\neg s_j^i$ by $DL1_j$. Since P_i is blocked, $p_j = <>$ and P_j must also be permanently blocked. But if $pp_j = <>$, P_j is not blocked by (2), whereas if $pp_j \neq <>$, P_j is not blocked by (3), so P_j cannot be permanently blocked and our assumption that P_i was blocked must be false.

6. Discussion

In [Mart80], A.J. Martin proposed a solution to this problem in terms of a *BROADCAST* operation (with which a node can send asynchronously and simultaneously along all output edges) and a *SELECT* operation, which chooses one input edge containing a message and reads that message. Messages on a given edge are received in the order sent. *BROADCAST* is delayed if sending along some edge would exceed the "slack bound" (buffering capacity) of that edge. *SELECT* is delayed if there is no pending message on any input edge. Martin proved that his algorithm requires a slack bound of at least two on each edge to avoid deadlock. Since our solution uses synchronous message passing, it requires a slack bound of zero. Martin's requirement for buffering seems to stem from the fundamentally asymmetric nature of *BROADCAST* and *SELECT*, since *BROADCAST* can send multiple messages, while *SELECT* can only receive one at a time.

Martin's algorithm is reproduced below (with minor notation changes):

```
if i=S → p, r, d, l := <S>, <S>, D, S; BROADCAST(p,d)
[] i≠S → r,l:=<>,NIL
fi;
k:=0;  {K = |R_i|}
do k < K → SELECT(p,d);
        if last(p) ≠ d ∧ l = NIL → p := p·i; l, r := i, p;
                                   BROADCAST(p,d)
        [] last(p) = d ∧ l ≠ d → l := d; k := k+1; r := p;
                                   BROADCAST(p,d)
        [] last(p) = d ∧ l = d → k := k+1
        [] last(p) ≠ d ∧ l ≠ NIL → skip
        fi
od
```

At termination, r holds the path from S to D. As p will hold the same path, it seems that r is mainly needed for the proof.

Our algorithm can be modified to use *BROADCAST* and *SELECT* operations, with only minor changes to its structure. However, there does not seem to be any easy way to modify Martin's algorithm to use a slack bound of zero.

Our algorithm, like Martin's, has a worst-case complexity of $O(E)$ messages sent (one partial path and one complete path sent along each link). The best-case complexity is about the same (anywhere between E and $2E$ messages sent, depending on the network), since a complete path must be sent along all links in addition to the partial path sent from S to D.

Most published distributed path algorithms deal with networks that allow bidirectional communication along any edge. While this is a reasonable model of wired communication systems, it is not always valid in other types, such as certain radio networks. An algorithm that does not depend on bidirectional communication is thus potentially valuable.

Although the algorithm described in this paper is not long, it is far from trivial to prove it correct. The difficulty is justified, however, by the inherent complexity of distributed programs. Attempting to demonstrate correctness through test cases or informal arguments leaves too many possibilities for error. A formal proof, on the other hand, gives far greater confidence in the correctness of the program.

Acknowledgements

We wish to thank David Gries, Gary Levin, and Rick Schlichting for their detailed criticisms of earlier drafts of this paper.

References

[Hoar78] Hoare, C.A.R. Communicating Sequential Processes. *CACM* 21, 8, August 1978, 666-677.

[LeGr81] Levin, G.M. and Gries, D. A Proof Technique for Communicating Sequential Processes. *Acta Informatica* 15, 3, June 1981, 281-302.

[Mart80] Martin, A.J. A Distributed Path Algorithm and its Correctness Proof. Technical Report AJM21a, Philips Research Laboratories, Eindhoven, The Netherlands, March 1980.