Programming Languages
J. J. Horning* Editor

An Exercise in Proving Parallel Programs Correct

David Gries
Cornell University

A parallel program, Dijkstra's on-the-fly garbage collector, is proved correct using a proof method developed by Owicki. The fine degree of interleaving in this program makes it especially difficult to understand, and complicates the proof greatly. Difficulties with proving such parallel programs correct are discussed.

Key Words and Phrases: garbage collection, multiprocessing, program correctness for multiprocessing tasks

CR Categories: 4.32, 4.34, 4.35, 4.39, 5.24

* Note. This paper was submitted prior to the time that Horning became editor of the department, and editorial consideration was completed under the former editor, Ben Wegbreit.

This research was partially supported by the National Science Foundation under grant GJ-42512. Author's present address: Department of Computer Science, Cornell University, Ithaca, NY 14850.

1. Introduction

At the NATO International Summer School on Language Hierarchies and Interfaces, Marktoberdorf, 1975, Edsger W. Dijkstra presented an "on-the-fly" garbage collector. Dijkstra and his colleagues had tackled this problem "as one of the more challenging—and hopefully instructive—problems" in parallel programming. Indeed, the high degree of interleaving of the processors' actions made his solution, and the arguments about its correctness, difficult to understand. The major difficulty was the lack of necessary tools and the lack of any systematic method for understanding parallelism. Having recently worked with Susan Owicki on her thesis [3] on methods for proving properties of parallel programs, it struck me that with Owicki's techniques I could perhaps provide a better understanding of the program. With some help from Dijkstra and Tony Hoare, I was able to arrive at an outline of a proof of correctness of the garbage collector and present it a few days later at the Summer School.

A fully detailed, complete proof, however, took me much longer, partly because I was not adept enough yet at applying the techniques, but also because proving properties of parallel programs is so much harder than proving properties of sequential programs.

Owicki's proof techniques deserve further study, and this paper attempts to describe them and their use in the context of Dijkstra's garbage collector. Section 2 presents and discusses some of Owicki's proof techniques. Section 3 describes the garbage collection problem and gives the solution, along with an informal discussion of its correctness. Section 4 is devoted to more formally establishing its correctness.

Parallel programming *is* much harder than sequential programming. The reader might want to study Section 5, the conclusions, after looking at the solution but before reading its correctness proof, in order to fully understand the problems of parallelism.

2. Definition and Use of the Language

Let S be a statement and P and Q assertions about variables. In [2], Hoare introduces notation like $\{P\}\ S\ \{Q\}$ to informally mean: if P is true before execution of S, then Q will be true when S terminates. This is a statement of *partial correctness*; termination of S must be established by other means.

As another notational device, by

$$\frac{a, b, c}{d}$$

we mean that if a, b, and c hold, then so does d. Hoare introduces axioms and inference rules similar to the following for a fragment of ALGOL.

null $\{P\}$ **skip** $\{P\}$ (2.1)

assignment $\{P^x_e\}\ x := e\ \{P\}$ (2.2)

where P_e^x represents the result of substituting (e) for each free occurrence of x in P; e.g. if P is $(a > 0 \wedge b = 1)$, then P_{a+b}^a is $(a + b > 0 \wedge b = 1)$. It should be recognized that this axiom holds only when variable x has no other name used in e or P; otherwise the axiom may not be consistent.

alternation
$$\frac{\{P \wedge B\}\ S1\ \{Q\},\ \{P \wedge \neg B\}\ S2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q\}} \tag{2.3}$$

iteration
$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{P \wedge \neg B\}} \tag{2.4}$$

composition
$$\frac{\{P0\}\ S1\ \{P1\},\ \{P1\}\ S2\ \{P2\},\ \ldots,\ \{P_{n-1}\}\ Sn\ \{Pn\}}{\{P0\}\ \textbf{begin}\ S1;\ \ldots;\ Sn\ \textbf{end}\ \{Pn\}} \tag{2.5}$$

consequence
$$\frac{\{P1\}\ S\ \{Q1\},\ P \vdash P1,\ Q1 \vdash Q}{\{P\}\ S\ \{Q\}} \tag{2.6}$$

Let us now briefly discuss proofs of properties of sequential programs. When we write $\{P\}\ S\ \{Q\}$, this implies the existence of a proof $\{P\}\ S\ \{Q\}$ using the axioms and inference rules (2.1)–(2.6). For example suppose we have already proved $\{P1 \wedge e\}\ S1\ \{Q1\}$, $\{P1 \wedge \neg e\}\ S2\ \{Q1\}$, $P \vdash P1^{x}_{a}$, and $Q1 \vdash Q$, and suppose we have

$S \equiv$ **begin** $x := a$; **if** e **then** $S1$ **else** $S2$ **end**

Then a proof of $\{P\}\ S\ \{q\}$ *might* be

(1) $\{P1^x_a\}\ x := a\ \{P1\}$ assignment

(2) $$\frac{\{P1^x_a\}\ x := a\ \{P1\},\ P \vdash P1^x_a}{\{P\}\ x := a\ \{P1\}}$$ consequence

(3) $$\frac{\{P1 \wedge e\}\ S1\ \{Q1\}\ \{P1 \wedge \neg e\}\ S2\ \{Q1\}}{\{P1\}\ \textbf{if}\ e\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q1\}}$$ alternation

(4) $$\frac{\{P1\}\ \textbf{if}\ e\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q1\},\ Q1 \vdash Q}{\{P1\}\ \textbf{if}\ e\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q\}}$$ consequence

(5) $$\frac{\{P\}\ x := a\ \{P1\},\ \{P1\}\ \textbf{if}\ e\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q\}}{\{P\}\ \textbf{begin}\ x := a;\ \textbf{if}\ e\ \textbf{then}\ S1\ \textbf{else}\ S2\ \textbf{end}\ \{Q\}}$$ composition

This proof can be outlined more compactly and understandably by interleaving statements and assertions:

```
{P}
begin {P}
      {P1^x_a}
      x := a;
      {P1}
      if e  then {P1 ∧ e} S1 {Q1}
           else {P1 ∧ e} S2 {Q1}
      {Q1}
      {Q}
end
{Q}
```
(2.7)

In a proof outline, two adjacent assertions $\{P1\}\ \{P2\}$ denote the use of the rule of consequence, where $P1 \vdash P2$. Second, each statement S is preceded directly by an assertion called the *precondition* of S, written $pre(S)$. The precondition of $x := a$ above is $P1^x_a$.

Owicki [3] introduces two statements for parallel processing. The **cobegin** statement indicates that processes are to be executed in parallel; the **await** statement provides synchronization and mutual exclusion. The **await** statement has the form

await B **then** S

where B is a Boolean expression and S a statement. Execution of the process is delayed at the **await** until B is true. At this time, S is executed as an indivisible operation—no other process may execute while S is executing or during the time that B is found to be true and execution of S is begun since this might falsify B. If two delayed processes have their corresponding Booleans B come true at the same time, one of them is further delayed while the other executes. The scheduling algorithm for determining which process is allowed to proceed does not concern us here. For simplicity, we assume that **await**s cannot be nested.

The formal definition of the **await** statement is:

await
$$\frac{\{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \textbf{await}\ B\ \textbf{then}\ S\ \{Q\}} \tag{2.8}$$

Before introducing the **cobegin** statement, let us explain what it means for two parallel processes to be *interference-free*.

Definition. Given $\{P\}\ S\ \{Q\}$, let T be any assignment or **await** (not in S) with precondition $pre(T)$. We say that T *does not interfere with the proof of* $\{P\}\ S\ \{Q\}$ if (a) $\{Q \wedge pre(T)\}\ T\ \{Q\}$, (b) for each statement S' of S which is not within an **await**, $\{pre(S') \wedge pre(T)\}\ T\ \{pre(S')\}$.

Thus execution of T cannot affect the truth of the preconditions and result conditions used in the proof of S, and hence the proof $\{P\}\ S\ \{Q\}$ holds even if T is executed while S is executing.

Definition. $\{P1\}\ S1\ \{Q1\}$ and $\{P2\}\ S2\ \{Q2\}$ are *interference-free* if each assignment statement of $S2$ (which does not occur within an **await**) and each **await** of $S2$ does not interfere with the proof of $\{P1\}\ S1\ \{Q1\}$, and vice versa.

If $S1$ and $S2$ are interference-free as just defined, then execution of $S2$ leaves valid all the arguments used in the proof $\{P1\}\ S1\ \{Q1\}$, and therefore the proof still holds in the face of parallel execution. This allows us to define the **cobegin** statement as follows:

parallelism
$$\frac{\{P1\}\ S1\ \{Q1\},\ \{P2\}\ S2\ \{Q2\}\ \text{interference-free}}{\{P1 \wedge P2\}\ \textbf{cobegin}\ S1 \parallel S2\ \textbf{coend}\ \{Q1 \wedge Q2\}} \tag{2.9}$$

Again, for simplicity, we assume a program has only one **cobegin** statement.

In any operational model consistent with this and the other axioms, statements $S1$ and $S2$ can be executed concurrently, and execution of a **cobegin** terminates

only when both $S1$ and $S2$ have terminated. No assumptions about the relative speeds of processes $S1$ and $S2$ are made. Evaluation of any expression or execution of any assignment, however, must be performed as an indivisible operation which cannot be interrupted, but we can lift even this restriction if we adhere to the following (which this paper does):

> Any expression e in process Si may contain at most one reference to at most one variable changed in the other process Sj. If variable x of $x := e$ in Si is referenced by process Sj, then e can contain *no* references to x or to a variable changed in Sj. (2.10)

For example, suppose process $S1$ changes variable a. Then process $S2$ may not contain the statements $a := a + 1$ or $b := a + a + 1$. If process $S1$ references a, then in process $S2$ an assignment $a := a + 1$ must be written as $t := a + 1$, $a := t$ where t is local to $S2$. The same restrictions hold for an array, where we consider an assignment $a[i] := e$ to be a change of the whole array a (as in the Pascal axiomatic definition [5]).

With this convention, the only indivisible action need be the memory reference. Suppose process Si changes variable (location) A while process $Sj, j \neq i$, is referencing A. The memory must have the property that the value received for A by process Sj is the value of A either before or after the assignment, but it may not be garbage caused by fluctuation of the state of memory during the assignment to A. Thus the methods described here can be used to prove properties of programs executing on any reasonable machine, with as fine a grain of interleaving as one could imagine. Dijkstra's on-the-fly garbage collector takes advantage of such a fine grain of interleaving.

One often must be able to delete variables from (or add variables to) a program in order to effect a proof. The following definitions allow this.

Definition. Let AV be a set of variables which appear only in assignments $x := e$ in a program S, where $x \in AV$. Then AV is an *auxiliary variable set for* S.

Definition. Let AV be an auxiliary variable set for S'. S is *a reduction of* S' if it is obtained from S' by one of the operations: (a) Delete all assignments $x := e$ where $x \in AV$, or (b) replace **await true then** $x := e$ by $x := e$, provided $x := e$ satisfies (2.10).

Auxiliary variable axiom. Let AV be an auxiliary variable set for S', S a reduction of S' with respect to AV, and P and Q assertions which do not contain variables from AV. Then

$$\frac{\{P\}\ S'\ \{Q\}}{\{P\}\ S\ \{Q\}}$$

That one can delete auxiliary variables in this manner should be obvious; their values do not affect the results of computation at all, but are used only in proving correctness.

We now have a system for proving partial correctness of parallel programs. We shall see that we cannot use it completely and formally because the processes we deal with may not even terminate. But we can use the insight gained to informally prove properties of parallel programs.

The formalization teaches us to understand parallel processes in two steps. First, prove the properties of each parallel process $S1$ and $S2$ as sequential programs, disregarding parallel execution. Second, show that execution of $S2$ does not destroy the *proof* of $S1$'s properties, and vice versa, for if parallel execution of $S2$ does not invalidate the proof, it cannot destroy the desired properties.

This is an important step forward in understanding parallelism. Earlier work has often tried to show that execution of $S2$ does not interfere with the *execution* of $S1$. By concentrating more on the *proof*, we turn our attention to a more static object which is easier to handle. Of course, the sequential proofs may turn out to be harder because we must often weaken or change the arguments so that they will not be destroyed by parallel activity. This will become clear later.

We shall subsequently apply this technique. We shall not prove that subparts of a sequential program work correctly if it is obvious; we use proof outlines as in (2.7), and we often leave out implications $P \vdash Q$ if they can be discerned by the reader. We also use other statement notations where clearer and make program transformations without a formal proof rule if the transformations are obviously correct. The assertions themselves will often be at a high, informal level in an attempt to be clear without having to resort to too much formalism.

3. On-the-Fly Garbage Collection

The data structure used in a conventional implementation of LISP is a directed graph in which each node has at most two outgoing edges (either of which may be missing): an outgoing left edge and an outgoing right edge. At any moment all nodes of the graph must be *reachable* (via a directed path along directed edges) from a fixed root which has a fixed, known place in memory. The storage allocated for each node is constant in size and can accomodate two pointers, one for each outgoing edge. A special value **nil** denotes a missing edge. We allow the directed graph to have cycles.

For any reachable node, an outgoing edge may be deleted, changed, or added. Deletion and change may turn formerly reachable nodes into unreachable nodes which can no longer be used by the program (henceforth called the *mutator*). These unreachable nodes are therefore called garbage. Nodes not being used by the mutator are stored on a *free list* maintained as a singly linked list. The mutator may delete the first node from the free list and insert it into the directed graph by placing an edge to it from a reachable node.

If the free list becomes empty, computation halts and a process called "garbage collection" is invoked.

Beginning with the root, all reachable nodes are marked; upon completion of this marking phase, all unmarked nodes are known to be garbage and are appended to the free list. Computation then resumes.

A major disadvantage of this arrangement is the unpredictability of the garbage collection interludes. Dijkstra and his colleagues therefore investigated the use of a second processor, the *collector*, which would collect garbage on a more continuous basis concurrently with the action of the mutator. The constraints imposed on their solution were:

> the "interference between collector and mutator should be minimal ..., the overhead on the activity of the mutator (as required for cooperation) should be kept as small as possible, and, finally, the ongoing activity of the mutator should not impair the collector's ability to identify garbage as such as soon as possible."

Their solution satisfies these criteria, and we make no improvement on it at all; we are concerned only with the description and proof of their solution. Overhead on the mutator are one or two simple assignments per changed or added edge, the only indivisible action need be the memory reference, and the only synchronization occurs when the mutator must wait for the free list to have more than one node before taking a node from it.

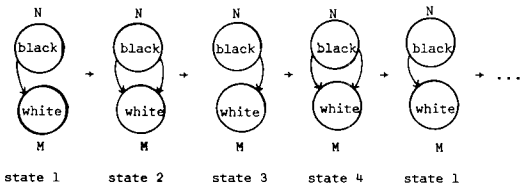
We now turn to the algorithm itself. The collector has two phases: *marking* reachable nodes and *collecting* unmarked, unreachable nodes. For marking, we must use three colors: *white* represents unmarked, *black* marked, and *gray* an "in-between" color needed for cooperation between collector and mutator. To see the need for the third color, suppose we use only black and white, and let nodes N and M be as depicted in state 1 of Figure 1. Now let the mutator repeatedly perform the following sequence of actions: insert a right-outgoing edge from node N to node M; delete the left-outgoing edge of node N; insert a left-outgoing edge from node N to node M; delete the right-outgoing edge of node N. M is thus always reachable from N. If M is white, the collector must notice that M is N's successor and blacken M. But the collector might never see this, for it might always check N's left-outgoing edge when it is **nil** (i.e. in state 3), and might always check N's right-outgoing edge when it is **nil** (i.e. in state 1). Thus the mutator must cooperate in some fashion and does so by graying a white node when it draws an edge to it.[1]

We now come to the representation of the graph of nodes. We use an array $m[0{:}N]$ for the nodes. **nil** *is represented by* 0, and thus the mutator itself may never reference node $m[0]$. This is not necessary, but makes presentation of the collector easier. We shall often speak of "node i" or just "i" instead of using the longer term "node $m[i]$."

Each node has three subfields which are of interest:

[1] It is possible to write a mutator-collector system using only the colors black and white [6]. We use the current system as a more interesting problem in parallelism.

Fig. 1. Noncooperation when using only two colors.



$$\begin{array}{ll} m[i].color & \text{current color of node (white, gray, or black)} \\ m[i].left & \text{node } i\text{'s left successor (0 if none)} \\ m[i].right & \text{node } i\text{'s right successor (0 if none)} \end{array} \tag{3.1}$$

The following *indivisible* actions are used to color nodes:

$$\begin{array}{ll} whiten(i): & m[i].color := white \\ blacken(i): & m[i].color := black \\ atleastgray(i): & \textbf{if } m[i].color = white \textbf{ then } m[i].color := gray \end{array} \tag{3.2}$$

Note that a black node is not made gray by operation *atleastgray*. These operations could be implemented with two bits, with white = 00, gray = 01, and black = 11. The operation *atleastgray*(i) would consist of "oring" the pattern 01 into $m[i].color$. It is important that this be performed with a single, indivisible, "or to memory" instruction, which is found on many machines. The possible sequence $t := m[i].color$ **or** 01; $m[i].color := t$ would violate the indivisibility requirement.

Two nodes $m[ROOT]$ and $m[FREE]$ are in fixed, constant places in the array $m[0{:}N]$. $m[ROOT]$ is the single root of the mutator's graph, while $m[FREE]$ is used to indicate where the free list begins. Within the collector, we consider $m[ROOT]$, $m[FREE]$, and $m[0]$ all to be *roots*; thus the free list and node $m[0]$ will be marked and unmarked just as the mutator's graph is.

The free list is maintained by using an extra integer variable *ENDFREE* to mark the end of the free list. Figure 2 illustrates the free list, while the following definition *Ifree* describes it more exactly; *Ifree* is invariantly true throughout execution.

Note that $m[FREE]$ is *not* a node of the free list, while $m[ENDFREE]$ is. Term (c) of *Ifree* covers the case that a node has been added to the free list but *ENDFREE* has not yet been readjusted. Such considerations, which may be inconsequential for sequential programs, are extremely important for parallelism.

Ifree $\equiv$

(a) the free list contains $j \geq 1$ nodes with distinct indices
$m[FREE].left = m[FREE].left^1 \neq 0$ $(0 = \textbf{nil})$,
$m[m[FREE].left].left = m[FREE].left^2 \neq 0$,
$\cdots$
$m[FREE].left^j \neq 0$; (3.3)

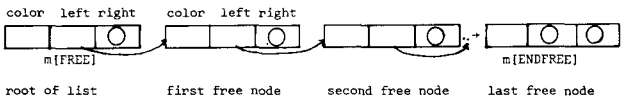
(b) $m[FREE].left^{j+1} = 0$;

(c) $ENDFREE = m[FREE].left^{j-1} \vee ENDFREE = m[FREE].left^j$;

(d) all nodes on the free list have no right successors.

[2] Steele [7] mentions that one need only *atleastgray*(j) in these procedures if the Collector is currently marking, but not when it is collecting. However, testing the state of the Collector and then perhaps *atleastgray*ing would only increase the complexity and decrease the speed of the system.

Fig. 2. The free list.



The mutator has at its disposal two procedures to add edges from one node to another[2]:

```
Add a left-outgoing edge from node k to node j:
proc addleft(k, j); begin m[k].left := j; atleastgray(j) end;
Add a right-outgoing edge from node k to node j:
proc addright(k, j); begin m[k].right := j; atleastgray(j) end
```
(3.4)

The mutator appears in (3.5) as a never-ending, nondeterministic guarded command loop [4]. Whether a guard on one of the guarded commands of the loop is really *true* on a given cycle of the outer loop is a function of the particular mutator being used; we assume for proof purposes that the guard could be true any time, and thus write *true*.

The two inner loops in the mutator are used to make it wait until the free list has two or more nodes before taking one off it. Variables k, j, and f are local to the mutator.

```
mutator: do true ⇒
  Let k, j be indices of nodes reachable from ROOT (k ≠ 0, j ≠ 0);
  if true  ⇒ m[k].left := 0
  [] true ⇒ m[k].right := 0
  [] true ⇒ addleft(k, j)
  [] true ⇒ addright(k, j)
  [] true ⇒ Take first free node as k's left successor:
            f := m[FREE].left;
            addleft(k, f);
            do f = ENDFREE ⇒ skip od;
            addleft(FREE, m[f].left);
            m[f].left := 0
  [] true ⇒ Take first free node as k's right successor:
            f := m[FREE].left;
            addright(k, f);
            do f = ENDFREE ⇒ skip od;
            addleft(FREE, m[f].left);
            m[f].left := 0
  fi
od
```
(3.5)

The collector is given below in (3.6). When first studying it, remember the insight gained from the formalism and treat it as an independent, sequential program under no parallel influence.

At the beginning of each execution of the body of the collector's main loop, there are no black nodes. Execution of the first section of the body grays the roots; so any reachable white node is reachable from a gray node (without going through a black node). After execution of the second section labeled *Blacken* (we look at this in detail subsequently), all reachable nodes are black; so all white nodes are garbage. The third section labeled *Collect* then searches through the nodes, appending white ones to the free list and whitening black nodes, in preparation for the next iteration.

```
Collector: do true ⇒
  Make roots at least gray:
    atleastgray(ROOT); atleastgray(FREE); atleastgray(0);
  Blacken: Blacken gray nodes and nodes reachable from gray
  nodes:
    i := 0;
    do i ≤ N and m[i].color ≠ gray ⇒ i := i + 1
    [] i ≤ N and m[i].color = gray ⇒ atleastgray(m[i].left);[3]
                                     atleastgray(m[i].right);
                                     blacken(i);
                                     i := 0
    od;
  Collect: Put white nodes on free list and whiten black nodes:
    for i := 0 step 1 until N do
      if m[i].color = white ⇒ Append i to free list:
                              m[i].left := 0;
                              m[i].right := 0;
                              m[ENDFREE].left := i;
                              ENDFREE := i
      [] m[i].color = black ⇒ whiten(i)
      [] m[i].color = gray ⇒ skip
      fi
od
```
(3.6)

The second section labeled *Blacken* searches through all nodes, both reachable and unreachable ones. Upon encountering a gray node, it grays its successors (if white) and then blackens it. Thus every reachable white node is always reachable from a gray node, which we express as

$$i \text{ white and reachable} \Rightarrow \exists \text{ path } (k_1, \ldots, k_p, i) \text{ where } k_1 \text{ is gray and } k_2, \ldots, k_p \text{ are white} \tag{3.7}$$

The effect is that, beginning with the gray nodes, all reachable nodes are first grayed and then blackened in waves spreading out from the roots. If a gray node becomes unreachable (because of mutator interaction), it nevertheless is blackened, along with its successors.

Each time a gray node is found and blackened, the collector begins checking the nodes again from the beginning. If no gray node is found during a complete traversal, then all nodes are black or white. From (3.7) and the absence of gray nodes, we conclude that all reachable nodes are black and that all white nodes are garbage and can be collected.

The node traversal algorithm in *Blacken* has been made simple and inefficient in order to simplify the correctness proof. Any traversal algorithm can be used which makes a final pass through all nodes without finding a gray node; this last pass is necessary because of interaction with the mutator.

The collector, as an independent algorithm, doesn't do much. It alternately blackens all white nodes—including the free list—and then makes them white again. The free list and the directed graph are repeatedly marked and unmarked, and there is never any garbage to collect.

Let us now consider interaction with the mutator. Suppose the mutator takes a node M from the free list and grays it (by using M as the second argument in a call to procedure *addleft* or *addright*) and then deletes an edge so as to make M unreachable. Thereafter the mutator may not reference M until it has been put on

the free list. During the subsequent marking phase the collector blackens M, and during the following collecting phase it whitens M. During the *next* marking phase M remains white and unreachable; so the following collection finally appends M to the free list. Thus any reachable node is appended to the free list within *two* collecting phases after it becomes unreachable.

This discussion does not prove correctness, for it does not take into full account the mutator interaction. The main problem is with mutator interference when the collector is about to blacken node i. The collector seems to assume that i's successors are nonwhite. However, just before the operation *blacken*(i) the mutator might interrupt and change i's successor to be an already existing white node. Blackening i could then lead to a black-to-white edge and destruction of the important invariant (3.7). Invariant (3.7) is needed in order to be able to prove that all reachable nodes have been blackened, or marked. We must show that (3.7) always holds during marking even in the face of mutator interaction.

The reader might wish to think about the following problem in light of this discussion: should procedure *addleft*(k, j) be written as $m[i]$.*left* := j; *atleastgray*(j) or as *atleastgray*(j); $m[i]$.*left* := j?

4. Proof of Correctness of the Mutator-Collector System

The mutator and collector, as given in (3.5) and (3.6), do not terminate; hence there are no properties that must hold upon termination. By "proving the correctness" of the system, we mean establishing certain properties or assertions that hold whenever the collector reaches certain states. For example, after section *Blacken*, all reachable nodes must be black, while after execution of *Collect* all nodes which were previously white (and unreachable) must have been put on the free list. We establish these properties by giving proof outlines of the main program, the collector's marking phase, the collector's collection phase, and the mutator, in that order. We follow this with a discussion of the interference-free property. We use the following notation:

$$\begin{aligned}
&\mathcal{L}n && \equiv m[n].left \text{ (n's current left successor),}\\
&\mathcal{R}n && \equiv m,[n].right \text{ (n's current right successor),}\\
&reach(n) && \equiv \exists\, path(ROOT, \ldots, n) \text{ or } path(FREE, \ldots, n),\\
&reachR(n) && \equiv \exists\, path(ROOT, \ldots, n),\\
&gray\text{-}reachable(n) && \equiv \exists\, path(k_1, \ldots, k_p, n) \text{ where } k_1 \text{ is gray and } k_2, \ldots, k_p \text{ are white.}
\end{aligned} \tag{4.1}$$

By a path $(k_1, \ldots, k_n)$ we mean a succession of distinct nodes k_i such that there is an edge from node k_i to node k_{i+1}. We use two auxiliary variables. Variable *mark* indicates whether or not the collector is currently marking; it is referred to in the mutator's *assertions* but not in the mutator itself. Variable *add* indicates whether the mutator is currently executing in procedure *addleft* or *addright*. These variables are needed only for the proof; by virtue of the auxiliary variable axiom, they may be deleted without disturbing the correctness of the system.

The mutator has its set of reasonable states under which it can cooperate; we describe these in the following assertions:

$$\begin{aligned}
&Mfree && \equiv Ifree \wedge \text{no free list node is reachable from } ROOT\\
&Mgraph && \equiv add = 0 \wedge (mark \Rightarrow \text{there is no black-to-white edge})
\end{aligned} \tag{4.2}$$

The mutator actions are simple enough to understand without a formal proof outline, but we must use these assertions and a proof outline in order to show noninterference. The collector also has its set of reasonable states which concern the free list and the colors of nodes at various stages of execution. We group them here to facilitate referencing them later.

A brief explanation of assertion $C\mathcal{L}$ will illustrate the nature of such assertions in parallel programming. On the basis of just the collector, we would like to have $C\mathcal{L} \equiv$ "i's left successor is nonwhite." However, as mentioned earlier, execution of the mutator may falsify this by changing $m[i]$.*left* to point to a white node. Hence $C\mathcal{L}$ is weakened to indicate that if i's left successor is white, then the mutator is at a specific point of execution (defined by variable *add*), and that i's left successor is gray-reachable from some other node besides i.

$$\begin{aligned}
&Cfree && \equiv Ifree \wedge \mathcal{L}ENDFREE = 0 \wedge \mathcal{L}0 = \mathcal{R}0 = 0\\
&Cmark && \equiv mark \wedge ROOT, FREE \text{ and } 0 \text{ are nonwhite} \wedge (n \text{ white and reachable} \Rightarrow \text{gray-reachable}(n))\\
&Cm(i) && \equiv 0 \le i \le N + 1 \wedge (m[0{:}i-1] \text{ contains a gray node implies } m[i{:}N] \text{ contains a gray node})\\
&C\mathcal{L} && \equiv \mathcal{L}i \text{ white} \Rightarrow i = add \ne 0 \wedge \exists path(k_1, \ldots, k_p, \mathcal{L}i) \text{ with } k_1 \text{ gray, } k_2, \ldots, k_p \text{ white, and } k_1 = i \text{ implies } k_2 = \mathcal{R}i\\
&C\mathcal{R} && \equiv \mathcal{R}i \text{ white} \Rightarrow i = -add \ne 0 \wedge \exists path(k_1, \ldots, k_p, \mathcal{R}i) \text{ with } k_1 \text{ gray, } k_2, \ldots, k_p \text{ white, and } k_1 = i \text{ implies } k_2 = \mathcal{L}i\\
&Ccoll(i) && \equiv \neg mark \wedge (0 \le n \le i \Rightarrow n \text{ nonblack}) \wedge ((i < n \le N \wedge n \text{ white}) \Rightarrow n \text{ is not reachable})
\end{aligned} \tag{4.3}$$

We are finally ready to give the proof outlines of the various sections. Look upon each as a sequential program and the proof outlines should not be hard to understand. Difficulties arise only because assertions have been weakened in order to show noninterference later on.

4.1 Proof Outline for the Main Program

Note that there is no terminating condition; whether the system halts depends on the particular mutator being executed. We are interested only in proving properties which hold as long as the mutator is executing.

{*Mfree* $\wedge$ *Cfree* $\wedge$ $\neg$ *mark* $\wedge$ *add* = *0* $\wedge$ no black nodes}
cobegin {*Cfree* $\wedge$ $\neg$ *mark* $\wedge$ no black nodes}
Collector: **do true** $\Rightarrow$
{*Cfree* $\wedge$ $\neg$ *mark* $\wedge$ no black nodes}

```
        Make roots at least gray;
        Blacken: Blacken gray nodes and nodes reach-
        able from grays;                                   (4.1.1)
        {Cfree ∧ ¬ mark ∧ all white nodes are unreach-
        able}
        Collect: Put white nodes on free list and whiten
        black nodes
        {Cfree ∧ ¬ mark ∧ no black nodes}
        od
  //    {Mfree ∧ Mgraph}
        mutator
coend
```

4.2 Proof Outline for the Marking Phase

This phase consists of sections *Make roots at least gray* and *Blacken*; a proof outline is given in (4.2.2). One difficulty is showing that execution of *blacken*(i) does not destroy assertion *Cmark*. From assertion ($C\mathcal{R} \wedge C\mathcal{L}$) one can conclude (because of the relations between i and *add*) that at most one of i's successors is white. Moreover, *if*, say, $\mathcal{L}i$ is white, then it is gray-reachable from a node *other* than i, and hence blackening i will not destroy the assertion *Cmark*.

We must show that the loop of (4.2.2) terminates. Consider the following integer function f:

$$f = 3 \cdot N \cdot (\text{no. of white nodes}) + 2 \cdot N \cdot (\text{no. gray nodes}) + N + 1 - i. \qquad (4.2.1)$$

Each execution of the loop body reduces f by at least 1. Furthermore $f \geq 0$. Hence after a finite number of iterations the loop terminates.

```
{Cfree ∧ ¬ mark ∧ no black nodes}
Make roots Root, Free and nil at least gray;
{Cfree ∧ ¬ mark ∧ no black nodes ∧ roots (including nil) are
  nonwhite} mark := true;
{Cfree ∧ Cmark}
i := 0;
{Cfree ∧ Cmark ∧ Cm(i)}.
do i ≤ N ∧ m[i].color ≠ gray ⇒ {Cfree ∧ Cmark ∧          (4.2.2)
                                 Cm(i) ∧ i nongray}
                               {Cfree ∧ Cmark ∧
                                 Cm(i + 1)}
                               i := i + 1
                               {Cfree ∧ Cmark ∧ Cm(i)}
[] i ≤ N ∧ m[i].color = gray ⇒ {Cfree ∧ Cmark ∧ i gray}
                               atleastgray(m[i].left);
                               {Cfree ∧ Cmark ∧ i gray
                                 ∧ Li nonwhite}
                               {Cfree ∧ Cmark ∧ i gray
                                 ∧ CL}
                               atleastgray(m[i].right);
                               {Cfree ∧ Cmark ∧ i gray
                                 ∧ CL ∧ Ri nonwhite}
                               {Cfree ∧ Cmark ∧ i gray
                                 ∧ CL ∧ CR}
                               blacken(i);
                               {Cfree ∧ Cmark ∧
                                 i black}
                               i := 0
                               {Cfree ∧ Cmark ∧ Cm(i)}
  {Cfree ∧ Cmark ∧ Cm(i)}
od;
{Cfree ∧ Cmark ∧ Cm(N + 1)}
{Cfree ∧ Cmark ∧ no gray nodes}
{Cfree ∧ Cmark ∧ all white nodes are unreachable}
mark := false
{Cfree ∧ ¬ mark ∧ all white nodes are unreachable}
```

4.3 Proof Outline for the Collecting Phase

```
Collect:
{Cfree ∧ ¬ mark ∧ all white nodes are unreachable}
{Cfree ∧ Ccoll(−1)}
for i := 0 step 1 until N do
  {Cfree ∧ Ccoll(i − 1)}
  if m[i].color = white ⇒                                  (4.3.1)
    {Cfree ∧ Ccoll(i) ∧ ¬ reach(i)}
    m[i].left := 0; m[i].right := 0;
    {Cfree ∧ Ccoll(i) ∧ ¬ reach(i) ∧ Li = Ri = 0}
    m[ENDFREE].left := i;
    {Ifree ∧ Ccoll(i) ∧ LENDFREE = i ≠ 0 ∧ L0 = R0 = 0}
    ENDFREE := i
    {Cfree ∧ Ccoll(i) ∧ ENDFREE = i}
  [] m[i].color = black ⇒ {Cfree ∧ Ccoll(i − 1) ∧ i black}
                           whiten(i)
                           {Cfree ∧ Ccoll(i)}
  [] m[i].color = gray ⇒ skip {Cfree ∧ Ccoll(i)}
  fi
  {Cfree ∧ Ccoll(i)};
{Cfree ∧ Ccoll(N)}
{Cfree ∧ ¬ mark ∧ no black nodes}
```

4.4 Proof of Properties of the Mutator

We begin with two lemmas about procedures *addleft* and *addright*. The extra auxiliary variable *add* is needed later to show noninterference; by the auxiliary variable axiom, assignments to *add*, as well as the **awaits**, can be deleted. Implied in these lemmas is that these procedures have no effect except as stated.

Lemma 4.4.1.

{*Mgraph* ∧ *reach*(k) ∧ *reach*(j)}
addleft(k, j)
{*Mgraph* ∧ *reach*(k) ∧ $\mathcal{L}k = j$}

and

{*Mgraph* ∧ *reach*(k) ∧ *reach*(j)}
addright(k, j)
{*Mgraph* ∧ *reach*(k) ∧ $\mathcal{R}k = j$}

Proof outlines.

```
proc addleft(k, j);
  begin {Mgraph ∧ reach(k) ∧ reach(j)}
        await true then begin m[k].left := j; add := k end;
          { k = add ≠ 0 ∧ reach(k) ∧ Lk = j
          ∧ (mark ⇒ the only possible black-to-white edge is
            (k, Lk)) }
        await true then begin atleastgray(j); add := 0 end
        {Mgraph ∧ reach(k) ∧ Lk = j}
  end
proc addright(k, j);
  begin {Mgraph ∧ reach(k) ∧ reach(j)}
        await true then begin m[k].right := j; add := −k end;
          { k = −add ≠ 0 ∧ reach(k) ∧ Rk = j
          ∧ (mark ⇒ the only possible black-to-white edge is
            (k, Rk)) }
        await true then begin atleastgray(j); add := 0 end
        {Mgraph ∧ reach(k) ∧ Rk = j}
  end
```

We are now ready to give the proof outline of the mutator. We show only three operations—those dealing with the left successor of a node; the other three operations dealing with the right successor are similar.

When showing noninterference, we shall also deal only with these three operations.

```
{Mfree ∧ Mgraph}
mutator : do true ⇒
  {Mfree ∧ Mgraph}
  Let k, j be indices of nodes reachable from ROOT;
  {Mfree ∧ Mgraph ∧ reachR(k) ∧ reachR(j)}
  if true ⇒ {Mfree ∧ Mgraph ∧ reachR(k)}
            m[k].left := 0
            {Mfree ∧ Mgraph ∧ reachR(k) ∧ ℒk = 0}
  [] true ⇒ {Mfree ∧ Mgraph ∧ reachR(k) ∧ reachR(j)}
            addleft(k, j)
            {Mfree ∧ Mgraph ∧ reachR(k) ∧ ℒk = j}
  [] true ⇒ Take first free node as k's left successor:
              {Mfree ∧ Mgraph ∧ reachR(k)}
              f := M[FREE].left;
              {Mfree ∧ Mgraph ∧ reachR(k) ∧ ℒFREE = f ≠ 0}
              addleft(k, f);
              { Ifree ∧ Mgraph ∧ reachR(k) ∧
                ℒFree = ℒk = f ≠ 0 ∧
                every path from ROOT to free list uses
                edge (k, ℒ k) }
              do f = ENDFREE ⇒ skip od;
              { Ifree ∧ Mgraph ∧ reachR(k) ∧
                ℒFREE = ℒk = f ≠ 0 ∧
                every path from ROOT to free list uses
                edge (k, ℒ k) }
              addleft(FREE, m[f].left);
              { Ifree ∧ Mgraph ∧ reachR(k) ∧
                ℒFREE = ℒf ∧ ℒk = f ∧ ℛf = 0
                every path from ROOT to free list uses
                edge (f, ℒ f) }
              m[f].left := 0
              fi
              {Mfree ∧ Mgraph}
            od
```

4.5 Showing Noninterference

We must show that the precondition of each statement S of the collector cannot be falsified by execution of the mutator, and vice versa. We must also show that function f (see (4.2.1)) remains a decreasing function under parallel execution, in order to show that the marking phase of the collector still terminates. It will help to handle the assertions in separate classes: first, those assertions which deal only with the structure of the graph—like *Ifree* and *reachR*(*k*)—and secondly, those that also deal with the coloring of nodes.

Noninterference of assertions dealing only with structure. Note first that *Ifree* is true throughout execution of both processes. Now consider the collector (3.6). The only statements that change the graph structure occur in *Append i to free list*. Here, the successors of an unreachable node i are deleted and i is appended to the free list. Hence the collector changes successors only of unreachable nodes and the last free list node $m[ENDFREE]$.

On the other hand, the mutator changes successors only of reachable nodes and never of node *ENDFREE* or of the last free list node. The mutator and collector work with disjoint subsets of the nodes in this respect.

With this insight, we now scan the collector's proof outline and make a list of those assertions which are obviously not falsified by the mutator:

Ifree, *Cfree*, $\mathscr{L}0 = \mathscr{R}0 = 0$,
$\neg$ *reach*(*i*), $\neg$ *reach*(*i*) $\wedge$ $\mathscr{L}i = \mathscr{R}i = 0$, (4.5.1)
ENDFREE = *i*, $\mathscr{L}$*ENDFREE* = *i*, *mark*, $\neg$ *mark*

In the same manner, we list the mutator assertions about structure which are not falsified by the collector:

Ifree, *Mfree*, assertions dealing with the reachability of nodes and who their successors are, such as *reach*(*k*), *reachR*(*k*) $\wedge$ $\mathscr{L}k = f \wedge \mathscr{L}f = \mathscr{R}f = 0$ (4.5.2)

We are able to handle the noninterference of these assertions informally because the sets of nodes which each process can work with (with respect to graph structure) are disjoint.

Noninterference of the other mutator preconditions. Three other mutator assertions must not be falsified by the collector, all dealing with the existence of black-to-white edges:

Mgraph ≡ *add* = 0 $\wedge$ (*mark* ⇒ there is no black-to-white edge),
k = *add* ≠ 0 $\wedge$ (*mark* ⇒ only possible black-to-white edge is (k, $\mathscr{L}k$), (4.5.3)
k = −*add* ≠ 0 $\wedge$ (*mark* ⇒ only possible black-to-white edge is (k, $\mathscr{R}k$).

We shall deal only with the first two. A scan of the collector yields the following three assignments, with relevant preconditions, which might falsify one of these assertions:

{no black nodes} *mark* := **true**
{$\neg$ *mark*} *white*(*i*) (4.5.4)
{*Cmark* $\wedge$ *C*$\mathscr{L}$ $\wedge$ *C*$\mathscr{R}$} *blacken*(*i*)

The only assignment where noninterference is not obvious is the third: *blacken*(*i*). Consider assertion *Mgraph*. We have {*Mgraph* $\wedge$ *Cmark* $\wedge$ *C*$\mathscr{L}$ $\wedge$ *C*$\mathscr{R}$} ⇒ {$\mathscr{L}i$ and $\mathscr{R}i$ are nonwhite}. Hence blackening node i under these conditions leaves *Mgraph* true. Next, consider the second assertion of (4.5.3). From this assertion and the precondition of *blacken*(*i*), we can prove that $\mathscr{R}i$ is nonwhite. Furthermore, if $\mathscr{L}i$ is white, then $i = k$ and the edge (i, $\mathscr{L}i$) is the same edge as (k, $\mathscr{L}k$). Thus blackening i leaves this assertion true.

Noninterference of the other collector preconditions. We note that the only coloring action of the mutator is to gray a reachable white node. Hence the mutator cannot falsify the assertions: *n nonblack*, *n black*, *n nonwhite*, *n gray*, $\neg$ *reach*(*n*) $\wedge$ *n white* for any arbitrary node n. We now scan the collector and make a list of the remaining assertions which must not be interfered with (see (4.3) for definitions):

Cmark
Cmark $\wedge$ *C*$\mathscr{L}$ (and similarly *Cmark* $\wedge$ *C*$\mathscr{R}$)
Cm(*i*) (for $0 \le i \le N + 1$)
Ccoll(*i*) (for $-1 \le i \le N$; this includes the assertion "all white nodes are unreachable") (4.5.5)
f does not increase (see (4.2.1) for the definition of f)

We show the noninterference only of the two interesting ones: *Cmark* and *Cmark* $\wedge$ *C*$\mathscr{L}$. We note that (*Mgraph* $\wedge$ roots nonwhite $\wedge$ *mark*) ⇒ *Cmark*. Since the mutator does not whiten nodes, it can falsify

Cmark only by falsifying *Mgraph* also. *Mgraph* is false only in two places, in procedures *addleft* and *addright*. We consider only one case. Suppose then that execution of the first **await** of *addleft* leaves a black-to-white edge $(k, \mathscr{L}k)$, under conditions

$\{Mgraph \wedge reach(k) \wedge reach(j) \wedge k$ black $\wedge j$ white $\wedge Cmark\}$
await true then begin $m[k].left := j$; $add := k$ **end**

Consider a white, reachable node i before execution of the **await**. Since *Cmark* is true, there exists a path $(k_1, \ldots, k_p, i)$ where k_1 is gray and $k_2, \ldots, k_p$ are white. Since node k is black, execution of $m[k].left := j$ cannot destroy this path. Hence i remains gray-reachable after execution of the **await**. Thus *Cmark* is also true after execution of the **await**.

Now consider assertion $Cmark \wedge C\mathscr{L}$. If node $\mathscr{L}i$ mentioned in $C\mathscr{L}$ is white, then the mutator is currently between the two **await**s of procedure *addleft* because the mutator's variable $add > 0$. Execution of the second **await** grays node j, which in this case is node $\mathscr{L}i$, and hence $C\mathscr{L}$ remains true after execution of the second **await.**

On the other hand, suppose $Cmark \wedge C\mathscr{L}$ is true and that node $\mathscr{L}i$ is nonwhite. The mutator can falsify $Cmark \wedge C\mathscr{L}$ only by making i's left successor white, and it can do this only by changing i's left successor through execution of the first **await** statement in procedure *addleft*. For this to happen, the following must be true *before* execution of the **await**:

$\{Cmark \wedge C\mathscr{L} \wedge \mathscr{L}i$ nonwhite $\wedge reach(k) \wedge reach(j)$
$\wedge j$ white $\wedge i = k\}$

From this we can conclude that there is a path $(k_1, \ldots, k_p, j)$ where k_1 is gray and $k_2, \ldots, k_p$ are white. Further, if node i lies on this path, its successor on the path cannot be $\mathscr{L}i$, which is nonwhite, and must therefore be $\mathscr{R}i$. This path is not disturbed by execution of the **await** and is exactly the path described in $C\mathscr{L}$ as the necessary condition if $\mathscr{L}i$ is white. Hence $Cmark \wedge C\mathscr{L}$ is not falsified by execution of this statement either.

5. Concluding Remarks

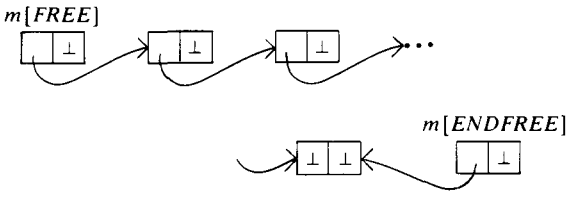
It has taken five and one half pages to introduce the topic, describe the proof techniques, give the solution, and describe it informally and another three to present a more detailed, formal proof of correctness. The complexity of parallel programs with such a fine degree of interleaving seems to be an order of magnitude greater than the complexity of corresponding sequential programs. While one might be able to improve on the style and presentation to make it appear simpler, such systematic methods must be used to master and control the complexity. In support of this view, I have thus far seen five purported solutions to this problem, either in print or ready to be submitted for publication. All used informal reasoning and had roughly the same mistake. Informal reasoning is just not adequate to handle the problem.

The main difficulty with the proof method used here is that assertions for one process have to be designed in the light of knowledge of the other processes. In the worst case, if one process has n statements and the other m statements, the proof method requires work proportional to $m \cdot n$. I suspect that no *general* proof method for parallel programs can improve on this worst-case bound.

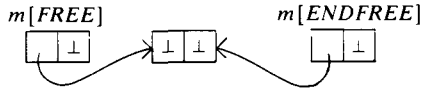
When we write a procedure to be used in a sequential setting, once it is written and proved correct we can view it as a black-box operation and use it over and over again without having to look in the black box. We worry only about *what* it does. In a parallel setting, however, we must analyze the procedure each time we wish to use it to make sure that the parallelism does not disturb its proof of correctness. And each change in the other process forces us to reanalyze the procedure again. One can avoid this complexity by making the procedure an indivisible operation through the use of synchronization and mutual exclusion primitives and by limiting the use of shared variables. Or one can summarize in an invariant for the procedure what a parallel process must leave true in order not to interfere.

The on-the-fly garbage collector is very fragile and susceptible to such changes. Slight changes which would seem innocent in a sequential setting are disastrous in a parallel context. Two examples of this would prove enlightening.

First of all, consider a free list where *ENDFREE* is handled as a fixed node not on the free list, as in (5.1). Diagram (5.2) shows a free list consisting of a single node.



(5.1)



(5.2)

Suppose the free list always contains at least one element. The mutator could use either of the following two waiting tests before taking a node from the free list:

$m[FREE].left \neq m[ENDFREE].left$ or
$m[m[FREE].left].left \neq 0$.

In sequential programming these might be equivalent, but in parallel programming the use of one or the other could lead to an error. The process which adds a node to the free list cannot change $\mathscr{L}ENDFREE$ and

$\mathcal{L}(\mathcal{L}ENDFREE)$ at the same time, so that only *one* of the following holds:

$$m[FREE].left \neq m[ENDFREE].left \Rightarrow m[m[FREE].left].left \neq 0.$$

or

$$m[m[FREE].left].left \neq 0 \Rightarrow m[FREE].left \neq m[ENDFREE].left.$$

Which one holds can only be determined by looking at the process which actually appends a node to the free list.

The second point illustrates the advantages of formal and systematic reasoning over informal reasoning. The last "bug" found in Dijkstra's program, which has appeared in four "solutions" that I have seen, was in the mutator's procedure *addleft*(k, j). This was first written as

atleastgray(j); $m[k].left := j$

Informal reasoning had led to the conclusion that j should be grayed first; so no black-to-white edge would exist. Only much later did informal reasoning by Mike Woodger find the error. Suppose the mutator is stopped after graying j. The collector begins marking and blackens k and j, begins collecting and whitens k and j, and then begins marking and blackens k. The collector pauses for a breath. The mutator executes $m[k].left := j$ and then deletes all other edges ending in j. The only way to reach j is through the black-to-white edge (k, j). At this point, the collector finishes marking, leaving j white, and then *collects* the white reachable node j, appending it to the free list.

Informal reasoning alone can never hope to think of all such torturous execution sequences; only a systematic method based on formal techniques can expect to cope with the complexity. With Owicki's techniques, the error would have been easy to find. In making up the list of mutator's assertions that may not be falsified we would have listed [*mark* $\Rightarrow j$ nonwhite]. Clearly execution of the collector's statement *mark* := **true** can falsify this assertion.

Having been warned of this error by Dijkstra, I cannot argue that I found and corrected it myself. But I venture to say that if I had used Owicki's methods carefully enough (one makes errors in proofs too), I would have found it easily. Of course, executing half of the sequence $m[k].left := j$; *atleastgray*(j) leaves a black-to-white edge momentarily, and it was Dijkstra who led me to use instead the invariant

{every reachable white node is reachable from a gray node}

in the collector. The assertion "there are no black-to-white edges" had to be replaced because of the mutator's actions.

Building a program with little regard to correctness and then debugging it to find errors is even more folly for parallel programs than it is for sequential programs. In either case, it is not sufficient. The goal of every programmer should be to make the detection of an error during testing the *exception* rather than the rule, as it is now. This means that the more complicated the problem and resulting program, the more systematically and formally the problem must be investigated in order to control the complexity.

I should add that the existence of a proof such as this does not imply absolutely that the program is correct; but it drastically raises our level of confidence in the program.

Acknowledgments. This exercise could not have been possible without a good knowledge of Owicki's thesis, and I am grateful for the privilege of supervising her thesis work. Thanks go to Dijkstra for showing us the garbage collection problem, to Dijkstra and Hoare for help in developing the proof in its various stages, to the members of IFIP working group 2.3 (programming methodology) for the opportunity to present and discuss this material at several meetings, and to the referees for their careful and constructive comments.

Originally submitted October 1975. Final revision submitted February 1977.

References

1. Dijkstra, E.W., et al. On-the-fly garbage collection: an exercise in cooperation. Notes for the 1975 NATO Summer School on Language Hierarchies and Interfaces, *Lecture Notes in Computer Science 46*, Springer-Verlag, New York, 1976.
2. Hoare, C.A.R. An axiomatic basis for computer programming. *Comm. ACM 12*, 10 (Oct. 1969), 576–583.
3. Owicki, S. Axiomatic proof techniques for parallel programs. Ph.D. Th., TR 75-251, Dept. Comptr. Sci., Cornell U., Ithaca, N.Y., July 1975.
4. Dijkstra, E.W. Guarded commands, nondeterminancy and formal derivation of programs. *Comm. ACM 18*, 8 (Aug. 1975), 453–457.
5. Hoare, C.A.R., and Wirth, N. An axiomatic definition of the programming language PASCAL. *Acta Informatica 2* (1973), 335–355.
6. Stenning, V. On-the-fly garbage collection. Unpublished notes, 1976.
7. Steele, G.L. Jr. Multiprocessing compactifying garbage collection. *Comm. ACM 18*, 9 (Sept. 1975), 495–508.