Dijkstra Book
Chapter on Concurrent Algorithms

Leslie Lamport

18 December 2021

Contents

1 Introduction

A concurrent algorithm is one in which concurrently executing, independent processes interact with one another. A parallel algorithm is one in which the processes are not independent, but were created to speed up execution of a single computation. There is no precise distinction between parallel and concurrent algorithms, and a concurrent algorithm may be used to synchronize the separate processes executing a parallel algorithm. However, they differ in practice and pose different problems. Distributed algorithms are a class of concurrent algorithms.

Dijkstra published five influential papers about concurrent algorithms. The first was the most important because it began the study of concurrent algorithms. This chapter describes most of the algorithms in those five papers, along with background material for some of them. Their descriptions here do not reflect how the algorithms were presented in the papers. I have extensively modified Dijkstra's notation and the way he described the algorithms, reflecting what has been learned in the decades since he did that work (and perhaps my biases). In particular, algorithms are written in the sort of informal pseudo-code I might have used in the 1980s and is still widely used today.

Dijkstra was careful to acknowledge colleagues who influenced his papers, often including them as coauthors. I believe he was the driving force behind these five papers, and I am reasonably sure that he did all the actual writing himself. I do not mean to belittle any of the coauthors of these papers (especially since I am one of them). However, for brevity, the papers are referred to simply as Dijkstra's.

2 Mutual Exclusion

In 1965, Dijkstra published in *Communications of the ACM* (CACM) a paper [5] containing what I believe was the first concurrent algorithm to appear in print. More important than the paper's algorithm was its introduction of the mutual exclusion problem that the algorithm solved. Even more important than that, the paper introduced the way we now think about concurrently executing processes. And it did all this in one page.

In the mutual exclusion problem, there are N processes, each of which may repeatedly execute a section of code called its *critical section*. The remainder of its code is called the *noncritical section*. A process may halt only in its noncritical section. The problem is to add synchronization code

to ensure that executions of the critical section by two different processes cannot occur at the same time. Dijkstra discussed this problem in EWD35, apparently written in 1962. He wrote there that, in 1959, the mathematician Theodorus Jozef Dekker found a solution for two processes, and that:

> [F]or almost three years, this solution has been considered a "curiosity", until these issues at the beginning of 1962 suddenly became relevant for me again...[1]

It was building the THE operating system [8] that made the problem relevant. That system was implemented as a collection of separate processes. Mutual exclusion was used, for example, to prevent two different processes from trying to print with the same printer at the same time.

2.1 Preliminaries

Dekker's and Dijkstra's algorithms, as well as many later mutual exclusion algorithms, are based on what I call the *one-bit protocol*: Each process has a flag (a Boolean-valued variable) that initially equals **true**; it enters the critical section by setting its flag to **false** and then waiting until it has read the value **true** in every other process's flag.[2] A process resets its flag to **true** upon exiting the critical section. No two processes can be executing their critical sections at the same time because the last one to set its flag to **false** would have read the other's flag equal to **false** if that other process were still in its critical section.

The one-bit protocol ensures mutual exclusion, but it does not solve the mutual exclusion problem. (I call it a protocol because it is used only as part of a complete algorithm.) If two processes both set their flags to **false** before reading the other process's flag, then something must be done to prevent both of them from waiting forever for the other to set its flag to **true**.

In Dekker's and Dijkstra's algorithms, this deadlock is broken by having one of the processes set its flag to **true** to let the other process enter the critical section before it does. This is also done in a simpler way by what I call the *one-bit algorithm*. This algorithm, for N processes numbered 1 through N, is described in Figure 1 using the following notation. The **variable** statement declares the shared variables, which can be accessed by

[1] From a translation by Martien van der Burgt and Heather Lawrence on the `cs.utexas.edu` web site.

[2] Following Dijkstra, I am using a flag whose value indicates if the process *does not* want to enter the critical section. These days, the protocol is described with a flag having the opposite meaning.

```
variable flag[i ∈ 1..N] = true
process i ∈ 1..N do
  while true
    do   noncritical section ;
      L: ⟨ flag[i] := false ⟩ ;
         for j ∈ 1..(i − 1)
           do if ⟨ ¬flag[j] ⟩ then ⟨ flag[i] := true ⟩ ;
                                   ⟨ await flag[j] ⟩ ;
                                   goto L
              fi
           od ;
         for j ∈ (i + 1)..N
           do ⟨ await flag[j] ⟩ od ;
         critical section ;
         ⟨ flag[i] := true ⟩
    od
od
```

Figure 1: The one-bit algorithm.

any process, and their initial values. The expression $i..j$ equals the set of all integers k with $i \leq k \leq j$. Variable *flag* is declared to be an array with index set $1..N$ such that *flag*$[i]$ equals **true** for all i in that set. The statement

process $i \in 1..N$ **do** ... **od**

declares that there is a process for each number in $1..N$, where "..." is the code for process number i. The meanings of the **while**...**do**...**od**, **if**...**then**...**fi**, and **goto** statements[3] should be obvious; and := denotes assignment. Boolean negation is written ¬. The **for** and **await** statements and the angle brackets ⟨ ⟩ require explanation.

Execution of the statement

for $j \in S$ **do** T **od**

means executing T once for each value of j in the set S. The executions for different values of j can occur in any order. At that time, Dijkstra

[3]In 1965, Dijkstra had not yet begun his crusade against the **goto**, and he used it freely. Simplicity is crucial for avoiding errors in concurrent algorithms. A **goto** should be used if it simplifies the algorithm's description.

would use a conventional **for** statement or other looping construct to perform the executions of statement T in a specified order, even when the order of execution didn't matter. He would sometimes mention that the order was irrelevant, but often didn't—perhaps because he expected it to be obvious to the reader.

Dijkstra assumed that the execution of a concurrent algorithm can be described as a sequence of atomic actions, each performed by a single process. (I will usually eliminate the word "atomic".) Exactly how execution of an operation is broken down into separate actions doesn't matter for single-process algorithms. It does for concurrent algorithms. Suppose two processes concurrently execute the statement $x := x + 1$. This increments x by 2 if execution of the statement is a single action. There are many other possible outcomes if reading and setting each bit in the representation of x is a separate action. Writing $\langle\, x := x + 1 \,\rangle$ indicates that the execution is a single action. For an expression E, writing $\langle\, E \,\rangle$ indicates that evaluation of E is a single action. Execution of an operation not in angle brackets can be split in any way into a sequence of actions.

The statement $\langle\, \textbf{await } E \,\rangle$ can be interpreted in two ways. One way is that the statement is executed only when the expression E equals **true**, and its execution just causes control to pass to the following statement. The other way is that it is equivalent to:

$$L : \langle\, \textbf{if } \neg E \textbf{ then goto } L \textbf{ fi} \,\rangle$$

which is the way Dijkstra wrote it (without the angle brackets). With the first interpretation, its execution consists of a single action. The second interpretation allows that action to be preceded by a sequence of actions that have no effect. An action that has no effect is unobservable, so those two interpretations are equivalent.

Let us now examine the algorithm. To enter its critical section, process i must set $flag[i]$ to **false** and then execute the two **for** loops. Completing execution of those loops requires reading $flag[j]$ equal to **true** for every other process j. The algorithm therefore implements the one-bit protocol for entering the critical section, ensuring mutual exclusion.

The algorithm also satisfies a property called *deadlock freedom*[4]: whenever some process is trying to enter its critical section, eventually (then or at a later time) some process is in its critical section. The proof is by contra-

[4]This name is misleading, because deadlock usually means that processes are waiting forever at **await** statements. Here, deadlock freedom also rules out livelock, where processes keep executing statements but never make progress.

diction. We assume that some process is trying to enter its critical section but no process is ever in its critical section, and we obtain a contradiction.

1. Eventually, every process is forever either in its noncritical section, or waiting at an **await** statement, or looping through statement L.

   PROOF: By the code and the assumption that no process ever again enters the critical section.

2. Eventually, every process is forever either in its noncritical section or waiting at an **await** statement.

   PROOF: Since no process is ever in its critical section, we need only show that no process is forever looping through statement L. If there is such a process, let process i be the lowest-numbered one. The code implies process i never examines the *flag* of any other process looping through statement L. By step 1, all the *flag* values it examines eventually never change, so it must eventually either exit its first **for** loop or wait forever at an **await** inside it. Hence, eventually no more processes can be looping through statement L.

3. Let process i be the lowest-numbered process with $flag[i]$ eventually always equal to **false**.

   PROOF: Such an i exists by step 2 and the assumption that some process is waiting to enter its critical section.

4. Contradiction.

   PROOF: By step 3, process i must be waiting forever for $flag[j]$ to equal **true**, for $j < i$. This implies i is in its first **for** loop, so $flag[i]$ equals **true**, contradicting step 3.

This algorithm was never mentioned by Dijkstra. It was discovered independently by James E. Burns, me, and perhaps others in the 1970s. To my knowledge, it was not published until the 1980s [3, 21]. However, it is so simple that it's hard to believe that Dijkstra, who understood the one-bit protocol, had not already discovered it by 1965. He might have, but then dismissed it for a reason discussed below.

2.2 Dekker's Algorithm

Deadlock freedom means that, if some process i is trying to enter the critical section, then some process j eventually enters it. It does not imply that i eventually enters it. The one-bit algorithm gives lower-numbered processes priority. A process may try forever to enter the critical section while lower-numbered processes keep entering and leaving. One might like the stronger

```
variables flag[i ∈ {0,1}] = true,  turn ∈ {0,1}
process i ∈ {0,1} do
  while true
    do      noncritical section ;
        L1: ⟨ flag[i] := false ⟩ ;
        L2: if ⟨ ¬flag[1 − i] ⟩
                then if ⟨ turn = i ⟩ then goto L2
                                    else  ⟨ flag[i] := true ⟩ ;
                                          ⟨ await turn = i ⟩ ;
                                          goto L1
                     fi
            fi ;
            critical section ;
            ⟨ turn := 1 − i ⟩ ;
            ⟨ flag[i] := true ⟩
    od
  od
```

Figure 2: Dekker's algorithm.

property that every trying process eventually enters the critical section. For a reason that should be obvious later, this property has come to be called *starvation freedom*.

Dekker's algorithm for two processes achieves starvation freedom by using a dynamic priority rather than the fixed priority based on process number of the one-bit algorithm. It has an additional variable *turn* whose value indicates which process has priority. If *turn* equals i, then process i waits with $flag[i]$ equal to **false**; otherwise, it waits with $flag[i]$ equal to **true**. Upon exiting the critical section, a process sets *turn* to the number of the other process. The algorithm, as it appeared in EWD123 (but with more modern notation), is in Figure 2. The processes are numbered 0 and 1, and the declaration of the variable *turn* indicates that its initial value can be either 0 or 1.

In the algorithm, processes execute the one-bit protocol to enter the critical section, guaranteeing mutual exclusion. Here is Dijkstra's proof of deadlock freedom, restated rather tersely in terms of our notation.

> Suppose both processes are trying to enter the critical section. If neither succeeds, then the value of *turn* remains constant. It is then easy to see that process number *turn* must eventually wait with $flag[turn]$ equal to **false** while process $1 - turn$ waits with

> *flag*$[1 - turn]$ equal to **true**, allowing process *turn* to enter the critical section. If only a single process i is trying to enter, then process $1 - i$ must reach the noncritical section with *flag*$[1 - i]$ equal to **true**, allowing process i to reach the critical section.

This argument is incomplete. It fails to consider the possibility that process i reads *flag*$[1 - i]$ equal to **false** in the outer **if** test and then waits forever with *turn* equal to $1 - i$ while process $1 - i$ remains forever in its noncritical section. It's not difficult to show that this can't happen, but the proof is not trivial.

Dijkstra's proof was based on considering all possible behaviors of the algorithm. The lacuna in the proof illustrates that this kind of reasoning is dangerous; it is hard to think of all the possible behaviors of a concurrent algorithm. While Dekker's algorithm is correct, we will see that such reasoning would later lead Dijkstra to write a correctness proof of an incorrect algorithm.

Although not asserted by Dijkstra, Dekker's algorithm is starvation free as well as deadlock free. In more than one place, Dijkstra mentioned the difficulty of finding and proving the correctness of concurrent algorithms, and he urged the reader to try to solve a problem before reading his solution. In that spirit, I leave the proof that Dekker's algorithm is starvation free to the reader.

2.3 Dijkstra's Algorithm

In the N-process mutual exclusion algorithm of [5], Dijkstra uses Dekker's idea of a variable *turn* that determines which among competing processes should be the next to enter the critical section. Instead of having the value of *turn* set by a process upon exiting the critical section, a process trying to enter the critical section tries to set *turn* to its own number. It can do so only when *turn* is the number of a process currently in its noncritical section. For this purpose, the algorithm uses an additional variable *idle*, where *idle*$[i]$ equals **true** when process i is in its noncritical section. The code is in Figure 3. The variable *temp* is declared to be local to the process, each process having its own copy. The expression $(1\,..\,N) \setminus \{i\}$ equals the set of all integers from 1 through N except i.

The algorithm obeys the one-bit protocol, since a process enters the critical section only by completing the **else** clause without looping back to L. Mutual exclusion is therefore guaranteed. To show deadlock freedom, it suffices to assume some process is trying to enter the critical section and no

```
variables flag[i ∈ 1..N] = true, idle[i ∈ 1..N] = true,  turn ∈ 1..N
process i ∈ 1..N
variable temp ∈ 1..N
  do while true
       do     noncritical section ;
              ⟨ idle[i] := false ⟩ ;
          L:  if ⟨ turn ≠ i ⟩
                then ⟨ flag[i] := true ⟩ ;
                     ⟨ temp := turn ⟩ ;
                     if ⟨ idle[temp] ⟩ then ⟨ turn := i ⟩ fi ;
                     goto L
                else ⟨ flag[i] := false ⟩ ;
                     for j ∈ (1..N) \ {i}
                       do if ⟨ ¬flag[j] ⟩ then goto L fi od ;
              fi ;
              critical section ;
              ⟨ flag[i] := true ⟩ ;
              ⟨ idle[i] := true ⟩
       od
  od
```

Figure 3: Dijkstra's algorithm.

process ever succeeds, and to show that some process eventually does enter the critical section. With no process entering the critical section, eventually every process either remains forever in its critical section or keeps looping through the **if** test at L. Since $idle[j]$ is **true** if process j is in its noncritical section, if process number *turn* is not looping, then some looping process will set *turn* to its own number. At that point, *turn* must always equal the number of a looping process, so $idle[turn]$ always equals **false**. Each looping process can then set *turn* at most once, so eventually the value of *turn* stops changing. The value of $flag[j]$ will then eventually forever equal **true** for every process except process *turn*, and process *turn* will enter its critical section.

While it is deadlock free, Dijkstra's algorithm is not starvation free even for two processes. One process i can set *turn* equal to i and keep cycling through its noncritical and critical sections, while another process waits forever, always reading $idle[i]$ equal to **false**.

2.4 Semaphores

These mutual exclusion algorithms were of intellectual interest only. They were impractical for implementing mutual exclusion in the THE system. Instead, Dijkstra introduced the semaphore, described in EWD35. A binary semaphore *sem* is a special kind of variable, whose value initially equals 0, that can be accessed only by the two operations $P(sem)$ and $V(sem)$.[5] These operations are defined by:

$$P(sem)\text{:} \quad \left\langle \begin{array}{l} \textbf{await } sem > 0 \ ; \\ sem := sem - 1 \end{array} \right\rangle$$

$$V(sem)\text{:} \quad \langle \ sem := 1 \ \rangle$$

Thus, the value of a binary semaphore always equals either 0 or 1. As reported in EWD123, Carel Scholten suggested allowing semaphores whose value can be any natural number, where the $V(sem)$ operation is replaced by $\langle \, sem := sem + 1 \, \rangle$.

It is trivial to implement mutual exclusion with a binary semaphore *sem*. Simply precede the critical section with a $P(sem)$ statement and follow it with $V(sem)$. Semaphores can also be used in other ways to synchronize processes. Binary semaphores are still commonly used today. They are now called locks, and P and V are called *lock* and *unlock*.

[5] P and V are the first letters of Dutch words meaning *pass* and *release*.

When multiple processes are waiting to execute a $P(sem)$ operation, execution of a $V(sem)$ allows one of them to proceed. Dijkstra said nothing about which one will enter. His definition allows an individual process to wait forever if other processes keep executing $P(sem)$ operations. The simple mutual exclusion algorithm using a single semaphore is then deadlock free but not starvation free.

Allowing a process to wait forever on a $P(sem)$ operation may not be acceptable. A semaphore *sem* is said to be *fair* if every process waiting on a $P(sem)$ operation will eventually execute it, assuming every $P(sem)$ operation is followed by a corresponding $V(sem)$ operation. Fair semaphore implementations often guarantee something stronger than fairness, ensuring that no process waits too long to execute a $P(sem)$ operation. It would be surprising if that were not the case for semaphores in the THE system. For proving properties such as starvation freedom that assert something eventually happens, fairness is usually all that is required.

2.5 A Closer Look at the Problem

We have seen Dijkstra's solution of the mutual exclusion property. More important than his solution was the statement of the problem, including what could be assumed about the processes. He first stated the mutual exclusion requirement. He then assumed that processes communicated through what are now called atomic shared memory registers, accessed only with read and write operations. (Remember that he wrote an **await** statement as a waiting loop.) During an execution of the algorithm, the results of operations to shared memory by all the processes were assumed to be the same as if those operations were performed in some sequential order consistent with each process's code. He then stated four requirements for a solution. The last three were:

1. Nothing could be assumed about the relative execution speeds of the processes. Left implicit was the assumption that the speed was not zero—that is, each process kept executing instructions unless it executed a **halt** instruction.

2. Any process could halt in its noncritical section.

3. Deadlock freedom. In stating this assumption, he explicitly ruled out any algorithm in which *"After you. No, after you."* blocking, though highly improbable, could continue indefinitely.

The importance of each of these requirements is remarkable. Requirement 1 was probably suggested by the THE system, in which all processes shared a single processor, so one process might do nothing for a long time while other processes used the processor. However, Dijkstra realized this was not just an artifact of that system. He understood that the distinction between process and processor would exist even on multiprocessor machines, and that processes could represent devices with very different execution speeds, such as computers, printers, and humans. The assumption that each process keeps executing, today called *process fairness*, is now assumed for most concurrent algorithms

Requirement 2 ruled out simple solutions in which all processes cycle through the critical section—a process immediately exiting it if it has nothing critical to do.

The remarkable part of requirement 3 is not deadlock freedom, but that it should hold for every possible execution. Using the 1-bit protocol, it's not hard to find algorithms that ensure mutual exclusion but allow the kind of blocking the requirement forbids. Dijkstra probably thought some people would think those algorithms were correct because such blocking is unlikely to persist for long.

We now take it for granted that correctness means correctness of all executions. But this paper was published two years before Floyd's seminal paper [13] that effectively launched the modern field of program correctness. I don't know of any algorithm that was published before this one with a correctness proof. Even 14 years later, no indication of why an algorithm might be correct was required for the ACM to publish it [1]. Testing was the standard way of verifying correctness of an algorithm. But testing cannot show that every possible execution of an algorithm does what it is supposed to—especially a concurrent algorithm, where the absence of a bound on relative execution speeds of the processes usually allows infinitely many possible executions. In 1965, only a proof could do that.

Dijkstra's remaining requirement, the first one on his list, was:

> The solution must be symmetrical between the N computers; as a result we are not allowed to introduce a static priority.

I used to think that this was added as a "poor man's substitute" for starvation freedom. It meant that, although his solution didn't guarantee that any particular process ever entered the critical section, it ensured that every process had an equal chance of entering it. But in EWD35, written three years earlier, Dijkstra mentioned that Dekker's algorithm was symmetric but did not say that it was starvation free. He apparently felt this condition

was more important than starvation freedom. He could have known the one-bit algorithm and not mentioned it because it was asymmetric.

The first starvation free N-process mutual exclusion algorithm, which was also symmetric, was published eight months later by Donald Knuth [17]. It essentially uses Dekker's idea of modifying the one-bit algorithm so process priority is indicated by a variable *turn* that is set by a process when exiting the critical section. The process numbers, in order of decreasing priority, are $turn$, $turn - 1, \ldots, 1, N, N - 1, \ldots, turn + 1$.

Almost all mutual exclusion algorithms published since then have been starvation free. Symmetry no longer seems to be considered important. For example, the popular bakery algorithm [18] ensures first-come, first-served entry to the critical section—arguably a fairer guarantee than symmetry—but is not symmetric because of a static priority among processes that arrive concurrently.

2.6 The Dining Philosophers

The mutual exclusion problem has been generalized in various ways to allow more than one process to be in its critical section at the same time. One early example is when the critical section serves to protect some data. A process must not modify the data while another process is trying to read or modify it. However, multiple processes that only read the data can access it at the same time. This is called the *readers/writers* problem. Processes are partitioned into readers and writers, and the mutual exclusion criterion is that a writer cannot be in its critical section while any other process is in its critical section. Early solutions used semaphores [4].

The most popular variant of mutual exclusion is the Dining Philosophers problem. It appears in EWD198, where Dijkstra calls it the "Problem of the Dining Quintuple". EWD1000 recounts that he invented it as an examination problem for a course he gave in 1965, and that C.A.R. (Tony) Hoare renamed it the Dining Philosophers problem.

In this problem, five philosophers alternate between thinking and eating. The philosophers eat at a round table, where each is assigned a seat with a plate in front of him. They are served a "difficult kind of spaghetti" that must be eaten with two forks.[6] There are five forks, one between each pair of philosophers. A philosopher can use only the two forks next to him; and he eats when in his critical section. Since a fork can be used by only one

[6]The insular nature of European cuisine at the time is displayed by the philosophers' use of two forks rather than a pair of chopsticks.

philosopher at a time, the mutual exclusion condition is that two adjacent philosophers cannot be in their critical sections at the same time.

Dijkstra evidently regarded this as a concurrent programming problem rather than an algorithm problem. In [9], he derived a solution using a global semaphore that allows only one philosopher at a time to examine the state of his two neighbors and modify his own state. It was an exercise in practical programming, not in finding an interesting algorithm. Dijkstra then observed that his solution was probably not satisfactory because a philosopher could be prevented from ever eating by voracious philosophers on either side of him. He said that a solution should allow every hungry philosopher to eventually enter his critical eating section, making it "starvation free". He indicated how such a solution can be obtained, but did not describe it.

The dining philosophers became very popular, both as a programming problem and an algorithmic one. I never felt it merited as much attention as it received; I believed its popularity was due to the cute story with which it was presented. A number of years later, my colleagues and I did some work that I thought deserved to be well known. The example of the dining philosophers inspired me to explain it with a story involving traitorous generals [23]. That worked quite well.

3 Self-Stabilization

3.1 The Problem

Nine years passed before Dijkstra published his second paper about concurrent algorithms, which also appeared in CACM [10]. I believe it was the first paper devoted to fault tolerance in concurrent algorithms.

Dijkstra observed that correct functioning of a system can be achieved by ensuring that it is always in a "legitimate" state. A fault in the program or the computer can cause incorrect behavior by putting the system in an illegitimate state. The system could recover from the fault by correcting its state.

An obvious way for the system to correct its state is to periodically check it and reset an illegitimate state to a legitimate one. This is not easy in a multiprocess system, where each process can examine only part of the state. Dijkstra's idea was to have each process, acting independently, cause the system to eventually go from any state to a legitimate once.

He defined a *self-stabilizing* algorithm to be one that, when started in any possible state, would eventually reach a legitimate state and operate properly from then on. He decided to take as his goal to make self-stabilizing

what we would now call a token ring.[7] As formulated by Dijkstra, there are $N+1$ processes numbered 0 through N, arranged in a circle. Let $L(i)$ and $R(i)$ equal $(i-1) \bmod (N+1)$ and $(i+1) \bmod (N+1)$, respectively. We call processes $L(i)$ and $R(i)$ the left and right neighbor of process i. In a token ring, there should always be a single token that resides at one of the processes. When process number i has the token, it must pass it to its right-hand neighbor, process number $R(i)$.

Expressed more precisely, the problem is to find an algorithm and a Boolean-valued function $HasToken(i)$ of the state of process i such that process i can perform an action only when $HasToken(i)$ equals **true**. A legitimate state is one in which $HasToken(i)$ equals **true** for exactly one process i, in which case its action makes $HasToken(i)$ equal **false** and $HasToken(R(i))$ equal **true**. Self-stabilization means that started in any possible state, the algorithm must eventually reach a legitimate state.

3.2 A Coarse-Grained Algorithm

Dijkstra devised three different self-stabilizing token ring algorithms. I will discuss only the first. It has a single array variable S indexed by process number, where $S[i]$ is in $0\,..\,(K-1)$ for a positive integer K whose value is constrained below. A legitimate state is one in which there is some process number i such that, for every process number j:

$$S[j] = \begin{cases} S[0] & \text{if } \ 0 \le j \le i \\ (S[0]+1) \bmod K & \text{if } \ i < j \le N \end{cases}$$

For i equal to N, this means all the $S[j]$ are equal.

Dijkstra defined $HasToken$ by:

$$HasToken(i) \ = \ \begin{cases} S[i] \neq S[L(i)] & \text{if } \ i \neq 0 \\ S[i] = S[L(i)] & \text{if } \ i = 0 \end{cases}$$

Remember that $L(0)$ equals N, so $HasToken(0)$ equals $S[0] = S[N]$.

It is easy to check that, in any legitimate state, $HasToken(i)$ equals **true** for exactly one process i. It is also easy to check that a process i other than 0 can pass the token to process $R(i)$ only by setting $S[i]$ equal to $S[i-1]$. Process 0 can pass the token to process 1 by changing $S[0]$ to $(S[0]+1) \bmod K$. Each token-passing action starting in a legitimate state then produces a legitimate state.

[7] The paper states that one of its algorithms was used shortly after being discovered, suggesting that Dijkstra chose a token-ring algorithm because he had an application in mind.

```
variables S[i ∈ 0..N] ∈ 0..(K − 1)
process i ∈ 0..N do
  while true do
    ⟨ await HasToken(i) ;
      if i ≠ 0 then S[i] := S[i − 1]
               else S[0] := (S[0] + 1) mod K
      fi
    ⟩
  od
od
```

Figure 4: The coarse-grained algorithm.

Dijkstra begins with a coarse-grained algorithm in which the entire operation of passing the token from i to $R(i)$ is a single action. We've just seen that this algorithm correctly implements a token ring when started in a legitimate state. We now need to verify self-stabilization. To do this, for each process i, we allow the initial value of $S[i]$ to be any number in $0..(K-1)$ and we show that the algorithm eventually reaches a legitimate state. The precise algorithm is in Figure 4. The algorithm assumes $K \geq N$. Here is Dijkstra's proof of self-stabilization.

1. In any possible state, *HasToken*(i) is true for at least one process i.

   PROOF: If *HasToken*(i) is false for each $i > 0$, then $S[i-1] = S[i]$ for all $i > 0$. This implies $S[0] = S[N]$, which implies *HasToken*(0) is true because $L(0) = N$.

2. After an action of process i has occurred, another action of process i cannot occur until the value of $S[L(i)]$ has changed.

   PROOF: Executing the process i action makes *HasToken*(i) false. It can become true again only by $S[L(i)]$ changing.

3. Infinitely many actions of every process occur.

   PROOF: By step 1, it is always possible for an action of some process to occur, so infinitely many actions of some process must occur. Since the value of $S[j]$ is changed only by an action of process j, step 2 implies that if infinitely many actions of any process i occur, then infinitely many actions of $L(i)$ occur. Hence, infinitely many actions of any one process implies infinitely many actions of all processes.

4. Eventually, a process 0 action that sets $S[0]$ to 0 occurs.

   PROOF: By step 3, infinitely many process 0 actions occur, and each such

action increments $S[0]$ by 1 modulo K.

Define the color of each process according to the following rules:

- Initially all processes are white.
- The first process 0 action that sets $S[0]$ to 0 turns process 0 blue. (Such an action occurs by step 4.)
- Node $i > 0$ is set blue either if process i executes an action when node $L(i)$ is blue, or if $L(i)$ is set blue and $S[L(i)]$ then equals $S[i]$. (Thus a process action that sets itself blue may also set one or more processes to its right blue.)

5. After process 0 becomes blue, at most $N-1$ process N actions can occur before the action that turns N blue.

   PROOF: Because values of $S[i]$ travel from left to right between processes 0 and N, and a value moving from a blue node to a white node turns the white node blue, step 2 implies that the values that $S[N]$ can have before N becomes blue must be among the white nodes to its left. When process 0 first becomes blue, there are at most $N-1$ white nodes to the left of node N. Hence, at most $N-1$ process N actions can occur before N becomes blue.

6. Immediately after N becomes blue, the sequence $S[0], S[1], \ldots, S[N]$ is nonincreasing.

   PROOF: Values move left to right, and new values are created by process 0 actions that increment $S[0]$ by 1 modulo K. By 2 and 5, until N becomes blue, at most $N-1$ process N actions have occurred. Since $S[0]$ equals 0 when process 0 becomes blue, we can use the assumption $K \geq N$ to deduce from this that the value of $S[0]$ can be at most $K-1$. Therefore, up to and immediately after execution of the action that makes N blue, the sequence of values of blue nodes is nonincreasing.

7. The algorithm is eventually in a legitimate state.

   PROOF: By step 6, eventually $S[0], S[1], \ldots, S[N]$ is nonincreasing. It is easy to see that this must remain true until a process 0 step decreases $S[0]$, which by step 3 must happen and by definition of the process 0 action can happen only after $S[N]$ equals $S[0]$. But $S[N] = S[0]$ and the sequence of values $S[i]$ being nondecreasing implies that all the $S[i]$ are equal, which is a legitimate state.

3.3 A Finer-Grained Algorithm

Dijkstra next considered the finer-grained algorithm obtained by modifying the algorithm in Figure 4 to make the **await** statement and the **if** statement separate atomic actions—in other words, adding "⟩" before the ";" and "⟨" before the "**if**". That modification to the code in Figure 4 would be adequate to describe the algorithm starting from a legitimate state. However, self-stabilization also requires showing that a legitimate state is reached from any possible starting state. The state of the finer-grained algorithm consists not only of the value of the array S, but also the control state of each process—that is, whether the next action of a process is execution of the **await** statement or the **if** statement. The algorithm should be self-stabilizing if the algorithm is started with each process in either possible control state.

Writing such an algorithm in pseudo-code would be complicated without additional language features. (It is easy to do with more mathematical ways of describing algorithms.) So, we will make do with this informal description of the finer-grained algorithm.

The proof of self-stabilization is almost identical to the proof for the coarse-grained algorithm. That proof breaks down in step 7. The assertion that a process 0 step can change $S[0]$ only when $S[0]$ equals $S[N]$ is incorrect. Suppose K equals N. It is possible that $S[N]$ equaled $N-1$ immediately before process N became blue. At that time, $S[0]$ could have equaled $N-1$ and found its **await** condition to be true, moving control to its **if** statement. Process N could then have become blue with $S[N]$ being set to 0. Process 0 could then have set $S[N]$ to $(N-1+1) \bmod N$ (since K equals N), which equals 0, and $S[0], S[1], \ldots, S[N]$ could then be nonincreasing, invalidating the proof.

This problem can be corrected by assuming $K > N$, and indeed this assumption implies that the algorithm is self-stabilizing. However, I hope that my admonition in Section 2.2 about the unreliability of behavioral reasoning has made the reader skeptical of my assertion that this correction produces a correct proof. There are now more reliable ways of proving properties like self-stabilization. However, they are beyond the scope of this chapter. The proofs presented here reflect the ones Dijkstra wrote, which today are best considered informal explanations rather than rigorous proofs.

3.4 What Dijkstra Actually Wrote

Dijkstra first discussed self-stabilization in EWD391, where he presented the coarse- and finer-grained algorithms and their proofs discussed above.

However, he did not describe those two versions of the algorithm in terms of atomic actions, but in terms of "demons" that control what action gets performed next. His first algorithm was obtained by a single centralized demon, and the second by what he called a "distributed" demon.

It is easy to see that the centralized demon produces the algorithm of Figure 4. However, there seem to be two reasonable interpretations of what execution by a distributed demon could mean. The first is the finer-grained algorithm discussed above. The second is to consider that the token-advancing operation for process $i \neq 0$ is performed by reading $S[L(i)]$ just once, rather than once for evaluating the **await** and again for performing the assignment to $S[i]$. In that case, the finer-grained version should have had an additional variable local to each process that stores the value of $S[L(i)]$ read in the **await** statement for use in the **if** statement.

Fortunately, there is now an easy way to determine which one Dijkstra meant: using a model checker to verify correctness of the algorithms for small values of N and K. A model checker checks the correctness of all possible executions, usually of a small instance of an algorithm. Model checking shows that the version that reads $S[L(i)]$ just once is not self-stabilizing for $K = N+1 = 4$. I conjecture that it is self-stabilizing for large enough values of K, but I will leave the proof or disproof of this as an exercise for highly motivated readers.

The description in terms of demons was not just ambiguous. It seems to have confused Dijkstra himself. EWD391 contained a proof of the finer-grained algorithm based on arguing that executing the algorithm with the distributed demon was equivalent to executing it with the centralized demon. That argument led to a "proof" that the finer-grained algorithm is self-stabilizing if $K \geq N$. EWD392 contains an erratum stating that the algorithm with a distributed demon requires the assumption $K > N$, but says nothing about requiring any change to the proof. There is a note crediting Carel Scholten with pointing out the error, saying "[It] shows a serious flaw in my reasoning; I should have known better!"

The other two algorithms in the CACM paper and their proofs first appeared in EWD392 and EWD396. However, no proofs were included in the published version, which was only $1\frac{3}{4}$ pages long. The word "proof" does not even appear.[8] Dijkstra certainly recognized the need for correctness proofs of concurrent algorithms. Perhaps he omitted the proof because he felt that a shorter paper would have more impact.

[8]The submitted version contained a comment about Scholten having found a nice proof for one of the algorithms, but the comment was eliminated in the published version.

3.5 The Paper's Influence

The coarse-grained algorithm described above requires at least N states per process. The coarse-grained version of the second algorithm in the CACM paper requires only 4 states per process, and that of the third algorithm requires only 3 states. (The finer-grained versions of these algorithms require $N + 1$, 4, and 3 states, plus the extra control state that is not accessed by other processes.)

I regard these algorithms to be the most brilliant work Dijkstra ever did. I was amazed by their simplicity and their depth. I was also impressed by the significance of the CACM paper. It provided the first rigorous approach to fault-tolerance in concurrent systems—indeed, the first discussion of the problem.

Unfortunately, very few others understood the paper's significance. This was probably due to how little motivation the paper provided for self-stabilization. The original submission (EWD397) said only that "the appreciation is left as an exercise to the reader". Probably in response to referees' comments, Dijkstra added two introductory paragraphs, but they mostly clarified the problem and provided just a bit of motivation. As a result, that paper almost completely disappeared until 1983.

In 1983, I gave an invited talk at the *Principles of Distributed Computing* (PODC) conference. In it, I talked briefly about the CACM paper and said how brilliant and how important it was. This introduced self-stabilization to the PODC community and led to it becoming an active research topic. I am proud of the part I played in helping this paper receive the attention it deserved. A discussion of the work the paper spawned as well as an explanation of the 4-state algorithm are in Chapter **??** written by Ted Herman.

The CACM paper also had an important effect on me personally. I discovered a generalization of the 4-state algorithm to a tree of processes [19] and sent it to Dijkstra. This led to his sending me some of his EWDs, which led to my involvement in the paper discussed next.

4 On-the-fly Garbage Collection

It was probably in April of 1975 that I received from Dijkstra a copy of EWD492, *On-the fly garbage collection: an exercise in cooperation.* I wrote to him suggesting a small simplification, and about a month later I received a copy of EWD496 with the same title, listing me among its five authors. A much-revised version was published in CACM in November 1978 [12]. The

story of how EWD496 became the CACM paper is instructive.

4.1 The Problem

In 1975, garbage collection was relevant mainly for LISP programs. Today, it arises in managing a heap of objects, which is at the heart of implementing object-oriented programs. The value of a program variable can be a pointer to an object in the heap, and a heap object can contain pointers to other heap objects. A heap object is said to be *reachable* if and only if it is pointed to by a variable or a reachable heap object. Only reachable objects can ever be used by the program; non-reachable objects are said to be *garbage*. Garbage collection is the reclaiming of memory space occupied by garbage objects.

Traditionally, garbage collection has been done by pausing the program, collecting the garbage, and then restarting the program. An interactive program becomes unresponsive while the garbage is being collected. The paper addresses the problem of eliminating this interruption by having garbage collection performed continuously by a *collector* process that runs concurrently with the main program. To eliminate details that do not affect the basic concurrency problem, the paper makes the following simplifying assumptions:

- There is a fixed set of variables that can contain pointers to heap objects.
- The size of the heap is fixed.
- All objects occupy the same amount of memory and have the same number of pointers.

The first assumption allows us to replace pointer-containing variables with special *root* objects that are, by definition, always reachable. The other assumptions mean there is no need to consider object creation and destruction. We can assume a fixed set of objects. The main program obtains new objects from a portion of the heap called the *free list*, reachable from special roots. Obtaining a new object then becomes an ordinary operation of the main program on reachable heap objects. Garbage collection consists of making garbage objects reachable as part of the free list. Real programs can contain null pointers, which don't point to any object. They can be considered pointers to a special root object called *null*.

The result of these simplifications is that all operations performed by the main program that change the heap can be represented as instances of

one simple operation: setting a pointer of a reachable object to point to a reachable object.

I now switch to notation similar to that used in the paper. The state of the heap is described as a directed graph with nodes and edges. If a pointer in object A points to object B, we say that there is an edge from node A to node B. A node is reachable if and only if it is reachable from a root node, under the usual definition of reachability in a directed graph.

Let *Nodes* be the (fixed) set of all nodes. The heap is represented by a variable *heap* whose value is an array indexed by the set *Nodes*. We are assuming that all nodes have the same number of outgoing edges, so we can represent the outgoing edges from a node A as an array $heap[A].edges$ of nodes indexed by a fixed set *EdgeIds* of edge identifiers. We represent the main program by a process called the *mutator*, and we call the garbage collecting process the *collector*. The mutator executes a sequence of operations

$$heap[A].edges[e] := B$$

for some reachable nodes A and B and some e in *EdgeIds*. The collector executes a sequence of *collections*, each of which can add unreachable nodes to the free list, making them reachable.

To represent the programming operation of creating a new object and making it reachable by mutator operations, the collector and mutator must obey some protocol for adding nodes to and removing them from the free list. The paper points out that this can be viewed as an instance of producer/consumer synchronization, which had known solutions [20]. We ignore how the free list is implemented by simply assuming that the collector can add a node to the free list with an atomic action, making it reachable with all its edges pointing to already reachable nodes. A collection will consist of a *marking* phase in which reachable nodes are identified, followed by a *freeing* phase in which nodes found not to be reachable are put on the free list.

There are two correctness conditions required of a solution:

CC1. Every garbage node is eventually added to the free list.

CC2. The collector makes no change to the heap except by putting garbage nodes on the free list.

Condition CC1 should be strengthened to require that garbage is collected in a timely fashion. The paper's algorithm ensures that any node that is garbage at the beginning of the freeing phase will be added to the free list by the end of the following complete collection. I will not discuss the proof

of CC1 and will consider only CC2. Since there is no reason for the collector to modify the heap other than by adding nodes to the free list, the only nontrivial part of satisfying CC2 is ensuring that no reachable node is ever added to the free list.

4.2 A Solution

Dijkstra wanted to minimize any cost to the mutator required by the algorithm. To avoid synchronization costs, he wanted the only atomic operations to the heap to be reading or changing a single edge of a single node. With that restriction, it's easy to construct scenarios in which a node remains reachable even though the collector never sees any edge pointing to it. So, in addition to changing edges, the mutator must inform the collector in some way when it changes them. This is done by having a flag, $heap[A].white$, initially equal to **true**, for every node A. (The flag is not considered to be part of the heap.) In the solution originally submitted to CACM, the mutator's single operation consists of these two atomic actions, for some reachable nodes A and B and some edge identifier e:

$$\langle\, heap[B].white \;:=\; \textbf{false} \,\rangle\ ;$$
$$\langle\, heap[A].edges[e] \;:=\; B \,\rangle$$

Thus, the only added cost for the mutator is setting one bit during every pointer-changing operation. There is also a *black* flag for each node, used only by the collector. We call a node A black if $heap[A].black$ equals **true**. A non-black node is said to be white if $heap[A].white$ equals **true** and to be gray if it equals **false**.[9] Initially, no node is black; and the collector's algorithm ensures that a node can be black only when the collector is performing a collection.

The collector's marking phase is shown in Figure 5, where *Roots* is the set of root nodes and *temp* is a variable local to the collector. The phase begins with the collector setting the roots gray. In the body of the **for** A loop, if node A is gray, then the collector: (i) sets the *white* flag false for every node pointed to by A, making that node gray if it was white, otherwise leaving its color the same, (ii) makes node A black, and (iii) begins the **for** A loop again by going back to control point L. Thus, the marking phase ends when the collector examines all nodes without finding a gray one. This must

[9] The paper describes the algorithm in terms of these three colors, without specifying how they are encoded. I prefer using the two bits because it makes clear that setting a node black or non-black modifies data accessed only by the collector, and that a node is made non-white by setting a bit without having to read it.

```
    for R ∈ Roots do ⟨ heap[R].white := false ⟩ od;
L:  for A ∈ Nodes
      do if ¬(heap[A].white ∨ heap[A].black)
           then for e ∈ EdgeIds do ⟨ temp := heap[A].edges[e] ⟩ ;
                                   ⟨ heap[temp].white := false ⟩
                                od ;
                heap[A].black := true ;
                goto L
         fi
      od
```

Figure 5: The collector's marking phase.

eventually happen, because each execution of that loop makes one gray node black.

To prove CC1, we must show that no reachable node is ever put on the free list. This requires showing that, at the end of the marking phase, every white node is garbage. Here is a sketch of Dijkstra's proof.

1. While the collector is in the marking phase, no black node ever points to a white node.

   PROOF: When a node A is turned black, the *white* flag of all the nodes pointed to by its edges had just been set false. If the mutator changed one of those edges before the collector made A black, it must have first set to false the *white* flag of the node the edge currently points to.

2. While the collector is in the marking phase's **for** A loop, every reachable node is reachable by a path from a black or gray node.

   PROOF: Every reachable node is reachable from a root node, and every root node has been turned grey and must remain either gray or black.

3. Upon completion of the marking phase, there is no gray node.

   PROOF: Suppose there is a gray node upon completion. It must have been white and become gray after being examined in the **for** A loop. Consider when the first such node became gray. By steps 1 and 2, it must then have been pointed to by a gray node that had not yet been examined by the loop. This is impossible because, when the loop examined that gray node, the **for** A loop would have been restarted.

4. Upon completion of the marking phase, no white node is reachable.

   PROOF: If there were a white reachable node at that point, consider the one reachable by the shortest path from a root. By steps 2 and 3, it is

reachable by a path from a black node, contradicting step 1.

The proof, sketched here and presented in more detail in the paper submitted to CACM, convinced Dijkstra and his four coauthors. It is wrong, and the algorithm is incorrect. Here is a scenario that displays the error.

The mutator begins the operation of making an edge from node A point to node B, making B gray. The collector then performs a complete collection, leaving the reachable nodes A and B both white. It then starts another collection, reaching a point in which it has made A black but has not yet examined B or made it non-white. The mutator then makes the edge of A point to B. There is now an edge from a black to a white node, showing that step 1 of the alleged proof is false. I will leave it to the reader to complete the scenario to one in which the collector finishes the marking phase, leaving B white, and then adds it to the free list in the freeing phase.

4.3 Verification

That five computer scientists, including someone as careful as Dijkstra, were fooled by an incorrect proof indicated that we needed more reliable methods of reasoning about concurrent algorithms. Edward Ashcroft had recently developed one method [2].

The simplest semantic view of concurrent algorithms that formalizes how Dijkstra reasoned about them is that an execution of an algorithm is a sequence of states, and a correctness condition is a predicate on such sequences. The type of correctness property at the heart of virtually all rigorous correctness proofs is invariance, which asserts that a predicate on states is true for every state of every execution. For example, step 1 of the incorrect proof asserts that this predicate on states is invariant:

MP. If the collector is in a marking phase, then no edge from a black node points to a white node.

Ashcroft observed that we can prove that a state predicate P is invariant by verifying these two conditions:

1. P is true of every initial state.

2. Any atomic action of the program beginning in any state satisfying P produces a state that satisfies P.

That these conditions imply P is true of every state of every execution follows by mathematical induction on the number actions required to reach the state.

A predicate P satisfying these two conditions is called an inductive invariant. Not every invariant is inductive. For example, even if MP were an invariant, it could not be inductive because condition 2 is not true for a state with a black node A and a white node B in which the mutator is about to perform the action that makes an edge of A point to B. To prove a formula is invariant, we must usually find an inductive invariant that implies it. To prove the invariance of MP, the inductive invariant would have to imply that, if the mutator is in a state in which it could make an edge of a black node A point to node B, then B is not white. An attempt to find such an inductive invariant would have failed and would have led to the counterexample. This actually happened after the fact.

I learned that there was an error in the algorithm by receiving a copy of a letter Dijkstra had sent to the editor of CACM withdrawing the paper. The letter said that M. Woodger had found the error, but gave no indication of what the error was. Since Dijkstra's proof was so convincing, I thought that the error must be minor and easily corrected. So, I decided to write an inductive invariance proof, thinking that I would then find and correct the error. In about 15 minutes, trying to write the proof led me to the error.

I suspected that the algorithm could be fixed by changing the order in which two atomic actions were performed. I had no good reason to believe that would work, and I could see no informal argument to show that it did. However, I decided to go ahead and try to find and prove the correctness of a suitable inductive invariant implying that only garbage nodes are white in the freeing phase. It took me about two days of solid work, but I did it. I was then convinced that the resulting algorithm was correct, but I still had no intuitive understanding of why it was correct

Meanwhile, Dijkstra had found a different fix and had written the same kind of proof as before. There was no reason not to use his algorithm, and I sketched an inductive invariance proof of it. Given the evidence of the unreliability of his style of proof, I tried to get Dijkstra to put a rigorous inductive invariance proof in the paper. He was unwilling to do that, though he did agree to make his proof closer to an invariance proof, with less behavioral reasoning. Here are his comments on that, written in July 2000 in a letter to me:

> There were, of course, two issues at hand: (A) a witness showing that the problem of on-the-fly garbage collection with fine-grained interleaving could be solved, and (B) how to reason effectively about such artifacts. I am also certain that at the time all of us were aware of the distinction between the two issues. I

> remember very well my excitement when we convinced ourselves that it could be done at all; emotionally it was very similar to my first solutions to the problem of self-stabilization. Those I published without proofs! It was probably a period in my life that issue (A) in general was still very much in the foreground of my mind: showing solutions to problems whose solvability was not obvious at all. It was more or less my style. I had done it with the mutual exclusion [algorithm].

Following Ashcroft, a few methods were proposed for verifying the two conditions needed to prove inductive invariance, their differences being based on the way the algorithm was described. I used one I had just devised [20] in my proof. The one developed by Susan Owicki and David Gries [24] was the most popular method, probably because they described the algorithm with ordinary programming-language constructs. Gries later used it to prove the published algorithm [15]. His proof was essentially the same as the one I had sketched. He simplified things a bit by combining two atomic operations into one,[10] but it would have been easy to add the details needed to handle the actual algorithm.

4.4 The Algorithm and its Significance

The correct algorithm of the published paper is obtained from the incorrect one by a simple but counterintuitive change to the mutator. The first atomic action of the mutator's operation of making an edge of node A point to node B is changed so that, instead of setting to **false** the *white* flag of node B, it sets to **false** the *white* flag of the previous node to which the mutator made an edge point.

The paper provides a path of reasoning that leads to the algorithm and a correctness proof. It calls this proof an "informal justification which we do *not* regard as an adequate substitute for a formal correctness proof." In the text of a talk Dijkstra gave in 1984 [6], he wrote this about concurrent and distributed systems:

> I know of only one satisfactory way of reasoning about such systems: to prove that none of the atomic actions falsifies a special predicate, the so-called 'global invariant'. Once initialized, the global invariant will then be maintained by any interleaving of the actions.

[10] He mentioned this simplification in a footnote that CACM failed to print, though it did include the footnote number in the text.

This is the only place I know in which he explicitly discussed the concept of an inductive invariant.

Dijkstra was wise to decide that the important contribution of the paper was the algorithm, not its proof of correctness. The algorithm implements the sharing of data, the heap, by concurrently executed processes using only atomic read and write operations. It does not use mutual exclusion or any synchronization primitive such as a semaphore. This means that neither process ever has to wait for the other one (except if the mutator finds the free list empty, when waiting is unavoidable). Such an algorithm is now called *wait-free* [16]. This was the first non-trivial[11] wait-free implementation of a shared data structure ever published.

5 Termination Detection

The last two of the five concurrent algorithm papers to be examined each had an algorithm for detecting termination in distributed computations. As documented in Section 4.3, Dijkstra was interested in finding concurrent algorithms to solve problems for which it was not clear that a solution existed. I think it must have been clear to him that solutions to this termination detection problem existed. He was probably interested less in the algorithms than in deriving them. He presented the algorithms with derivations based on inventing the invariants that the algorithms should maintain. I suspect he "derived" the algorithms after inventing them. But our concern here is the algorithms, not how they might be derived. The algorithms are therefore presented without rigorous derivations.

5.1 The Problem

Assume a set of processes that collectively perform a terminating computation, communicating with one another by sending messages. The content of the messages and what is being computed are irrelevant. All that we care about is that a process can be in one of two states: *idle* or *active*. An active process can send messages to other processes. A process can change from idle to active only when it receives a message. An active process can become idle at any time. The computation terminates when all processes are idle.

One process is designated the *leader*. The problem is to superimpose on the computation an algorithm by which the leader can detect that the

[11] I apparently invented an earlier wait-free implementation of a bounded FIFO queue for use as an example in [20], but it seemed so trivial that I assumed it was already known.

computation has terminated. The algorithm must satisfy two properties:

DT1. The leader can detect that the computation has terminated only if it actually has terminated.

DT2. If the computation terminates, then the leader eventually detects that it has.

To implement the algorithm, the processes use additional *control* messages. Idle as well as active processes may be required to send and receive control messages. I call the two solutions Dijkstra presented the *tree algorithm* and the *ring algorithm*.

5.2 The Tree Algorithm

The tree algorithm [7] assumes that the processes form a connected directed graph, where process A can send messages to process B if and only if there is an edge from A to B. The leader is assumed to have only outgoing edges.[12] Initially, only the leader is active. The control messages are acknowledgments (acks), sent in the opposite direction along edges. It is assumed that no messages are ever lost, but there is no assumption that messages arrive in the order they are sent—not even those sent along a single edge.

Define a process to be in a *neutral* state if it is idle, it has received acks for every message it has sent, and it has sent acks for every message it has received. Initially, only the leader is active and no messages have been sent, so every process except the leader is in a neutral state. The algorithm ensures that after the computation has terminated, every process eventually enters a neutral state. Moreover, when the leader is in a neutral state, all other processes are also in a neutral state.

A process leaves a neutral state only by receiving a message. A process in a non-neutral state remembers on which edge it received that message. Let's call that edge the process's *up edge*. We require that a process acknowledge the receipt of every message it has received subject to one rule: it must leave one message received on its up edge unacknowledged until acknowledging it makes the process neutral. In other words, a process can send the one remaining unsent ack to its up process only when it is idle, it has received acks for every message it has sent, and it has sent acks for every other message it has received. It must eventually send that last ack when allowed by the rule.

[12] A leader that receives messages can be simulated by a pair of processes, one acting as a leader that sends only a single message to the second.

When process A sends a message to process B that makes the edge from A to B the up edge of B, process A can't become neutral until process B does, because only when B becomes neutral will it send the last ack that A is waiting for from that edge. This implies that the non-neutral processes form a tree with the leader as root, the parent of each non-neutral process B in the tree being the process that is the source of the up edge of B. This in turn implies that if the leader is neutral, then all processes are neutral, so the computation has terminated. Hence DT1 is satisfied, where the leader detects that the computation has terminated when it is in a neutral state.

Once the computation has terminated, all processes are idle and no more messages are sent or received, so all acks except the last one to each non-neutral process's parent will eventually be sent. At that point, any leaf of the tree of non-neutral processes can and eventually must send its last ack and become neutral. Thus, all processes, including the leader, must eventually become neutral, showing that DT2 is satisfied.

5.3 The Ring Algorithm

The ring algorithm [11] assumes that processes can be numbered from 0 through $N-1$ such that process number i can send control messages to process number $(i-1) \bmod N$. The leader is process 0. Control information for the algorithm is also attached to the computation's messages. The algorithm assumes that the computation's message passing is instantaneous. Thus, the atomic action in which process i sends a computation message (with its control information) to an idle process j also makes j active. The algorithm can start with any set of processes active.

As in many distributed algorithms, a process has no control state. Its "code" is a set of *action specifications*, each consisting of a code fragment describing possible atomic actions that can be performed when the code fragment is enabled. The main computation is performed by two kinds of atomic actions of each process i: a $SendMsg(i)$ action sends a message from i to another process that at the same time receives it, and a $GoIdle(i)$ action puts the process in the idle state.

Termination detection is performed by a series of *probes*, in which a token is passed from each process i to process $(i-1) \bmod N$, starting from process 0. When Process 0 has the token, it can initiate a probe by executing a *StartProbe* action that passes the token to process $N-1$. Each process $i > 0$ can perform a $PassTok(i)$ action that passes the token to process $i-1$. The probe ends when the token returns to process 0.

The ring algorithm has the form shown in Figure 6. The construct

process 0 **do**
  while true do
    $\langle\, GoIdle(0) \,\rangle$ **or** $\langle\, SendMsg(0) \,\rangle$ **or** $\langle\, StartProbe \,\rangle$
  od
od

process $i \in 1..(N-1)$ **do**
  while true do
    $\langle\, GoIdle(i) \,\rangle$ **or** $\langle\, SendMsg(i) \,\rangle$ **or** $\langle\, PassTok(i) \,\rangle$
  od
od

Figure 6: The ring algorithm.

$\langle\, A_1 \,\rangle$ **or** $\ldots$ **or** $\langle\, A_k \,\rangle$ indicates that any one of the atomic actions described by A_i can be performed if it is enabled. We now have to define the $GoIdle(i)$, $SendMsg(i)$, $StartProbe$, and $PassTok(i)$ actions.

We start by considering what happens if no messages are ever sent, so the only main computation action a process can perform is the $GoIdle(i)$ action. In that case, we can let the $PassTok(i)$ action be enabled if and only if process i is idle. After the token of a probe started by the leader returns to the leader, every other process is idle and the computation has then terminated when the leader is idle.

This algorithm obviously doesn't work when messages can be sent. After the token is relayed by an idle process, a message from another process can make that process active. When that happens, we allow the token to continue its trip back to the leader, but we ensure that the leader learns nothing from it and must start a new probe. We do this by adding a color to the token. The probe starts with the color white, and it is turned black to tell the leader that the probe has *failed* and it must start a new probe. Process i can make a probe fail by having the $PassTok(i)$ action make the token black. Once black, the token remains black for the rest of the probe.

The probe should fail if a process i sends a message to an idle process j that the token has passed. Process j can't make the token black because the token has already passed it, so process i must do it. Our algorithm wouldn't represent a distributed system if we expected process i to know, when it sends the message, whether process j is idle or where the token is. So, we must be conservative. When process i sends *any* message, we have it turn the token black the next time it passes the token. To let process i remember that it has sent a message, we add a bit of information that we

```
variables active[i ∈ 0..(N − 1)] ∈ {true, false} ,
          proclr[i ∈ 0..(N − 1)] ∈ {"white", "black"} ,
          tokloc ∈ 0..(N − 1) ,
          tokclr = "black"

actions GoIdle(i):    active[i] := false

        SendMsg(i):   await active[i] ;
                      with j ∈ (0..(N − 1)) \ {i} do active[j] := true od ;
                      proclr[i] := "black"

        StartProbe:   await    (tokloc = 0)
                            ∧ ((tokclr = "black") ∨ (proclr[0] = "black")) ;
                      tokclr := "white" ;
                      proclr[0] := "white" ;
                      tokloc := N − 1

        PassTok(i):   await (tokloc = i) ∧ active[i] ;
                      tokloc := i − 1 ;
                      if proclr[i] = "black" then tokclr := "black" ;
                                                  proclr[i] := "white"
                      fi
```

Figure 7: Action specifications of the ring algorithm.

call the process's color. The color of a process is initially white, and it is made black when the process sends a message. The probe should also fail if process 0 has sent a message, so it should fail if process 0 is black as well as if the token is black.

This line of reasoning has effectively led us to an algorithm. If this were 1983, we would now have to do some careful thinking to find out if that algorithm is correct. Today, such thinking is unnecessary if a model checker can find an execution that shows the algorithm to be incorrect. Instead of thinking, we can write exactly what the algorithm is and run a model checker on it.

A precise description of our algorithm is in Figure 7. The variables have the following meanings:

active[*i*] Equals **true** if process i is active and **false** if it is idle.

proclr[*i*] The color of process i, indicated by the string "white" or "black".

tokloc The number of the process that holds the token.

tokclr The color of the token.

These variables can have any possible initial value, except that *tokclr* must initially equal "black". Thus, if the initial state is one in which a probe is being executed, that probe will fail. Here is an explanation of the four kinds of actions:

GoIdle(i) Makes process i idle. The action has no effect (does not change the state) if it is executed when the process is already idle. As observed in Section 2.1, an action that has no effect is unobservable, so it makes no difference whether or not it is executed.

SendMsg(i) Enabled if and only if the **await** expression is true—that is, if process i is active. As before, $(0\,..\,(N-1)) \setminus \{i\}$ equals the set of all process numbers except i. The **with** statement executes its **do** statement for an arbitrary value of j in that set, making process j active. The action also sets the color of process i to black. (If i is already black and j already active, then the action has no effect.)

StartProbe Performed only by process 0, it is enabled when that process has the token and it or the token is black (or both are black). It makes the process's color and the token's color both white and passes the token to process $N-1$.

PassTok(i) Performed by a process $i \neq 0$, it is enabled when the process has the token and is idle. It passes the token to process $i-1$. If the process's color is black, it makes the token black and itself white. Otherwise, it leaves the process's color white and the token's color unchanged.

A probe is defined to have succeeded if process 0 has the token and is idle, and both process 0 and the token are white—more precisely, when this formula is true:

$$(tokloc = 0) \wedge \neg active[0] \wedge (tokclr = proclr[0] = \text{“white”})$$

Note that if process 0 and the token are white, then process 0 will not pass the token and the current probe will succeed when process 0 becomes idle.

Any general-purpose model checker should be able to check if the algorithm is correct for small values of N.[13] I would expect an incorrect

[13] A few model checkers cannot check properties like DT2 that assert something must eventually happen.

algorithm to be incorrect even for $N = 3$; and I would be surprised if it were not incorrect for $N = 4$. For $N = 8$, the ring algorithm has 983,806 reachable states and the model checker I used verified DT1 and DT2 in 1 minute, 20 seconds on my laptop.[14]

Having checked that the algorithm is almost certainly correct, I will explain *why* it is correct by writing a correctness proof. I will start by proving the difficult part, DT1, which requires proving that a probe cannot succeed if the computation has not terminated.

By definition of what it means for a probe to succeed, it suffices to assume that the token has reached process 0 and some process is active, and to prove that the token or process 0 is black. If this is the first time the token reached process 0, then it is black because it was initially black and only process 0 can turn it white. We can therefore assume the token has passed process 0 already, so we are at the end of a complete probe that started at process 0. Since a process passes the token only when it is idle, the existence of an active process means that some process must have become active after it passed the token. Let i be the first process to become active after passing the token. Let j be the process that had the token when i first became active, so $j < i$. Process i can have become active only by receiving a message. Since i was the first process to become active after being passed by the token, every process k with $j < k$ was then idle, so the message that made i active must have been sent by a process k with $k \leq j$. Sending the message made k black, and $k \leq j$ implies that k hasn't yet passed the token. This means that either the token must have been black after k passed it, or else $k = 0$ so process 0 is black. This finishes the proof of DT1.

An examination of the proof reveals that the algorithm remains correct if we modify the $SendMsg(i)$ action so it colors process i black only if it sends a message to a process j with $j > i$.

The proof of DT2 is simpler. We assume that all processes are idle and show that a probe must eventually succeed. Since all processes are idle, every process $i > 0$ will pass the token, so the token must eventually reach process 0. If the token and process 0 are both white, then the probe has succeeded. Otherwise, process 0 will start the next probe, which will turn all the processes white. If the token is white when it reaches process 0, that probe will have succeeded. If not, process 0 will then start another probe that must succeed because all the processes are idle and white.

[14] I checked a slightly modified version of a TLA$^+$ [22] specification of Dijkstra's algorithm written by Stephan Merz, and I also checked (by hand) that the definitions in Figure 7 were equivalent to the ones in his specification.

Dijkstra's experience with the garbage collection algorithm tells us that we shouldn't believe these proof sketches. Dijkstra's derivation is more rigorous because he wrote down the invariants implicit in the proof sketches. However, model checking for all values of N from 1 through 8 gives me more confidence in the algorithm's correctness than his derivation. (Neither Dijkstra's derivation nor my proof sketch considers the case $N = 1$.) But this chapter is about algorithms, not about rigorous proofs, so we must be content with a proof sketch.

5.4 A Bit of History

In his earlier concurrent algorithms, Dijkstra (and his collaborators) stood alone. No one else had even thought of the problems he was solving. He was also a pioneer in distributed termination detection, but he was not alone. Nissim Francez was independently working on the same problem [14] and cited an unpublished paper by Michel Sintzoff that was also about the problem.

Dijkstra was careful to cite any work that directly influenced a paper he was writing. For example, the tree algorithm paper cites a lecture by P. M. Merlin for inspiring the problem. However, he did not mention Francez's work, even though he should have been aware of it before [7] was published. It is not clear why he did not follow the usual practice of citing such related work. Perhaps he felt that his termination detection papers were about derivation, not about the algorithms, so Francez's work was not relevant to them.

6 Conclusion

The five papers discussed in this chapter are remarkable. The first initiated the field of concurrent algorithms. The second two each started an important subfield of concurrent algorithm research. The last two were "merely" significant. And concurrent algorithms were not Dijkstra's primary interest.

I learned a lot from Dijkstra, and he learned a little bit from me. By the early 1980s, I believe we both felt we had nothing more to learn from each other. I remained cognizant of the debt I and others owed him for his seminal work on concurrent algorithms. However, it was only after he died that I understood how much more we owed him.

There are a number of ideas that have been central to the study of concurrent algorithms—for example, representing the execution of an algorithm as a sequence of states. Many of those ideas were in his papers, sometimes

only implicitly. They seemed obvious to me, so I assumed they were well known. But some ideas become obvious only after someone has thought of them. I have come to realize that many of the ideas that we take for granted are due to Dijkstra. Even the idea that one should prove the correctness of algorithms, though not unique to him, was radical at the time.

I have a more personal debt to Dijkstra. Before it ever occurred to me that there could be a science of computing, he was teaching me how to be a computer scientist.

References

[1] ACM. ACM algorithms policy. *CACM*, 22(5):329–330, May 1979.

[2] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, February 1975.

[3] James E. Burns, Paul Jackson, Nancy A. Lynch, Michael J. Fischer, and Gary L. Peterson. Data requirements for implementation of N-process mutual exclusion using a single shared variable. *Journal of the ACM*, 29(1):183–205, January 1982.

[4] P. J. Courtois, F. Heymans, and David L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, October 1971.

[5] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[6] E. W. Dijkstra. Invariance and non-determinacy. *Phil. Trans. R. Soc. Lond.*, A312:491–499, 1984.

[7] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[8] Edsger W. Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[9] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[10] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[11] Edsger W. Dijkstra, W.H.J. Feijen, and A.J.M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, 1983.

[12] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[13] R. W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Math., Vol. 19*, pages 19–32. American Mathematical Society, 1967.

[14] Nissim Francez. On achieving distributed termination. In Gilles Kahn, editor, *Semantics of Concurrent Computation, Proceedings of the International Symposium, Evian, France, July 2-4, 1979*, volume 70 of *Lecture Notes in Computer Science*, pages 300–315. Springer, 1979.

[15] David Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921–930, December 1977.

[16] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[17] D. E. Knuth. Additional commments on a problem in concurrent program control. *Communications of the ACM*, 9(5):321–322, May 1966.

[18] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[19] Leslie Lamport. On self-stabilizing systems. Technical Report CA 7412-0511, Massachusetts Computer Associates, December 1974. `https://www.microsoft.com/en-us/research/publication/self-stabilizing-systems/`.

[20] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[21] Leslie Lamport. The mutual exclusion problem—part ii: Statement and solutions. *Journal of the ACM*, 32(1):327–348, January 1986.

[22] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2003. A link to an electronic copy can be found at `http://lamport.org`.

[23] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[24] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–284, May 1976.