The Computer Science of Concurrency: The Early Years

Leslie Lamport
Microsoft Research

28 February 2015

To Edsger Dijkstra

It is insufficiently considered that men more often require to be reminded than informed.
Samuel Johnson

Contents

1 Foreword

I don't know if concurrency is a science, but it is a field of computer science. What I call *concurrency* has gone by many names, including parallel computing, concurrent programming, and multiprogramming. I regard distributed computing to be part of the more general topic of concurrency. I also use the name *algorithm* for what were once usually called programs and were generally written in pseudo-code.

This is a personal view of the first dozen years of the history of the field of concurrency—a view from today, based on 40 years of hindsight. It reflects my biased perspective, so despite covering only the very beginning of what was then an esoteric field, it is far from complete. The geneses of my own contributions are described in comments in my publications web page.

The omission that would have seemed most striking to someone reading this history in 1977 is the absence of any discussion of programming languages. In the late 1960s and early 1970s, most papers considered to be about concurrency were about language constructs for concurrent programs. A problem such as mutual exclusion was considered to be solved by introducing a language construct that made its solution trivial. This article is not about concurrent programming; it is about concurrent algorithms and their underlying principles.

2 The Beginning: Mutual Exclusion

2.1 The Problem

While concurrent program execution had been considered for years, the computer science of concurrency began with Edsger Dijkstra's seminal 1965 paper that introduced the mutual exclusion problem [5]. He posed the problem of synchronizing N processes, each with a section of code called its *critical section*, so that the following properties are satisfied:

Mutual Exclusion No two critical sections are executed concurrently. (Like many problems in concurrency, the goal of mutual exclusion is to eliminate concurrency, allowing us to at least pretend that everything happens sequentially.)

Livelock Freedom If some process is waiting to execute its critical section, then some process will eventually execute its critical section.

Mutual exclusion is an example of what is now called a *safety* property, and livelock freedom is called a *liveness* property. Intuitively, a safety property asserts that something bad never happens; a liveness property asserts that something good must eventually happen. Safety and liveness were defined formally in 1985 [1].

Dijkstra required a solution to allow any computer to halt outside its critical section and associated synchronizing code. This is a crucial requirement that rules out simple, uninteresting solutions–for example, ones in which processes take turns entering their critical sections. The 1-buffer case of the producer-consumer synchronization algorithm given below essentially is such a solution for $N = 2$.

Dijkstra also permitted no real-time assumption. The only progress property that could be assumed was *process fairness*, which requires every process that hasn't halted to eventually take a step. In those days, concurrency was obtained by having multiple processes share a single processor. One process could execute thousands of steps while all other processors did nothing. Process fairness was all one could reasonably assume.

Dijkstra was aware from the beginning of how subtle concurrent algorithms are and how easy it is to get them wrong. He wrote a careful proof of his algorithm. The computational model implicit in his reasoning is that an execution is represented as a sequence of states, where a state consists of an assignment of values to the algorithm's variables plus other necessary information such as the control state of each process (what code it will execute next). I have found this to be the most generally useful model of computation—for example, it underlies a Turing machine. I like to call it the *standard model*.

The need for careful proofs should have become evident a few months later, when the second published mutual exclusion algorithm [9] was shown to be incorrect [10]. However, incorrect concurrent algorithms are still being published and will no doubt continue to be for a long time, despite modern tools for catching errors that require little effort—in particular, model checkers.

2.2 The First "Real" Solution

Although of little if any practical use, the bakery algorithm [11] has become a popular example of a mutual exclusion algorithm. It is based on a protocol sometimes used in retail shops: customers take successively numbered tickets from a machine, and the lowest-numbered waiting customer is served next. A literal implementation of this approach would require a

ticket-machine process that never halts, violating Dijkstra's requirements. Instead, an entering process computes its own ticket number by reading the numbers of all other synchronizing processes and choosing a number greater than any that it sees.

A problem with this algorithm is that ticket numbers can grow without bound. This shouldn't be a practical problem. If each process chooses a number at most one greater than one that was previously chosen, then numbers should remain well below 2^{128}. However, a ticket number might have to occupy more than one memory word, and it was generally assumed that a process could atomically read or write at most one word.

The proof of correctness of the algorithm revealed that the read or write of an entire number need not be atomic. The bakery algorithm is correct as long as reading a number returns the correct value if the number is not concurrently being written. It doesn't matter what value is returned by a read that overlaps a write. The algorithm is correct even if reading a number while it is changing from 9 to 10 obtains the value 2496.

This amazing property of the bakery algorithm means that it implements mutual exclusion without assuming that processes have mutually exclusive access to their ticket numbers. It was the first algorithm to implement mutual exclusion without assuming any lower-level mutual exclusion. In 1973, this was considered impossible [4, page 88]. Even in 1990, experts still thought it was impossible [21, question 28].

One problem remained: How can we maintain a reasonable bound on the values of ticket numbers if a read concurrent with a write could obtain any value? For example, what if reading a number while it changes from 9 to 10 can obtain the value 2^{2496}? A closely related problem is to implement a system clock that provides the current time in nanoseconds if reads and writes of only a single byte are atomic, where a read must return a time that was correct sometime during the read operation. Even trickier is to implement a cyclic clock. I recommend these problems as challenging exercises. Solutions have been published [12].

2.3 A Rigorous Proof of Mutual Exclusion

Previous correctness proofs were based on the standard model, in which an execution is represented as a sequence of states. This model assumes atomic transitions between states, so it doesn't provide a natural model of the bakery algorithm with its non-atomic reads and writes of numbers.

Before I discuss a more suitable model, consider the following conundrum. A fundamental problem of interprocess synchronization is to ensure

that an operation executed by one process precedes an operation executed by another process. For example, mutual exclusion requires that if two processes both execute their critical sections, then one of those operation executions precedes the other. Many modern multiprocessor computers provide a Memory Barrier (MB) instruction for implementing interprocess synchronization. Executing an instruction A then an MB then instruction B in a single process ensures that the execution of A precedes that of B. Here is the puzzle: An MB instruction enforces an ordering of two operations performed by the same process. Why is that useful for implementing interprocess synchronization, which requires ordering operations performed by different processes? The reader should contemplate this puzzle before reading the following description of the *two-arrow* model.

In the two-arrow model, an execution of the algorithm is represented by a set of *operation executions* that are considered to have a finite duration with starting and stopping times. The relations $\rightarrow$ and $\dashrightarrow$ on this set are defined as follows, for arbitrary operation executions A and B:

$A \rightarrow B$ is true iff (if and only if) A ends before B begins.

$A \dashrightarrow B$ is true iff A begins before B ends.

It is easy to check that these relations satisfy the following properties, for any operation executions A, B, C, and D:

A1. (a) $A \rightarrow B \rightarrow C$ implies $A \rightarrow C$ ($\rightarrow$ transitively closed)
(b) $A \not\rightarrow A$. ($\rightarrow$ irreflexive)

A2. $A \rightarrow B$ implies $A \dashrightarrow B$ and $B \not\dashrightarrow A$.

A3. $A \rightarrow B \dashrightarrow C$ or $A \dashrightarrow B \rightarrow C$ implies $A \dashrightarrow C$.

A4. $A \rightarrow B \dashrightarrow C \rightarrow D$ implies $A \rightarrow D$.

The model abstracts away the explicit concept of time and assumes only a set of operation executions and relations $\rightarrow$ and $\dashrightarrow$ on it satisfying A1–A4. (An additional property is needed to reason about liveness, which I ignore here.)

Proving correctness of the bakery algorithm requires some additional assumptions:

- All the operation executions within a single process are totally ordered by $\rightarrow$.

- For any read R and write W of the same variable, either $R \dashrightarrow W$ or $W \rightarrow R$ holds.

Each variable in the algorithm is written by only a single process, so all writes to that variable are ordered by $\rightarrow$. We assume that a read that doesn't overlap a write obtains the correct value. More precisely, if a read R of a variable satisfies $R \rightarrow W$ or $W \rightarrow R$ for every write W of the variable, then R obtains the value written by the latest write W with $W \rightarrow R$.

With these assumptions, the two-arrow formalism provides the most elegant proof of the bakery algorithm that I know of. Such a proof of a variant of the algorithm appears in [14].

The conundrum of the MB command described at the beginning of this section is easily explained in terms of the two-arrow formalism. Suppose we want to ensure that an operation execution A in process p precedes an operation execution D in a different process q—that is, to ensure $A \rightarrow D$. Interprocess communication by accessing shared registers can reveal only that an operation execution C in q sees the effect of an operation execution B in p, which implies $B \dashrightarrow C$. The only way to deduce a $\rightarrow$ relation from a $\dashrightarrow$ relation is with A4. It allows us to deduce $A \rightarrow D$ from $B \dashrightarrow C$ if $A \rightarrow B$ and $C \rightarrow D$. The latter two $\rightarrow$ relations can be ensured by using MB instructions, which enforces $\rightarrow$ relations between operation executions by the same process.

3 Producer-Consumer Synchronization

3.1 The FIFO Queue

The second fundamental concurrent programming problem to be studied was producer-consumer synchronization. This form of synchronization was used at the hardware level in the earliest computers, but it was first identified as a concurrency problem by Dijkstra in 1965, though not published in this formulation until 1968 [6]. Here, I consider an equivalent form of the problem: a bounded FIFO (first-in-first-out) queue. It can be described as an algorithm that reads inputs into an N-element buffer and then outputs them. The algorithm uses three variables:

in The infinite sequence of unread input values.

buf A buffer that can hold up to N values.

out The sequence of values output so far.

```
--algorithm PC {
    variables in = Input,  out = ⟨⟩,  buf = ⟨⟩ ;
    fair process (Producer = 0) {
      P: while (TRUE) {
           await Len(buf) < N ;
           buf  := Append(buf, Head(in)) ;
           in   := Tail(in)                    } }
    fair process (Consumer = 1) {
      C: while (TRUE) {
           await Len(buf) > 0 ;
           out := Append(out, Head(buf)) ;
           buf := Tail(buf)                    } }   }
```

Figure 1: Producer-consumer synchronization

$$\left[\begin{array}{l} in = \langle v_1, v_2, \ldots\rangle \\ out = \langle\,\rangle \\ buf = \langle\,\rangle \end{array}\right] \xrightarrow{P} \left[\begin{array}{l} in = \langle v_2, v_3, \ldots\rangle \\ out = \langle\,\rangle \\ buf = \langle v_1\rangle \end{array}\right] \xrightarrow{C}$$

$$\left[\begin{array}{l} in = \langle v_2, v_3, \ldots\rangle \\ out = \langle v_1\rangle \\ buf = \langle\,\rangle \end{array}\right] \xrightarrow{P} \left[\begin{array}{l} in = \langle v_3, v_4, \ldots\rangle \\ out = \langle v_1\rangle \\ buf = \langle v_2\rangle \end{array}\right] \xrightarrow{P} \cdots$$

Figure 2: An execution of the FIFO queue.

A *Producer* process moves values from *in* to *buf*, and a *Consumer* process moves them from *buf* to *out*. In 1965 the algorithm would have been written in pseudo-code. Today, we can write it in the PlusCal algorithm language [15] as algorithm *PC* of Figure 1. The initial value of the variable *in* is the constant *Input*, which is assumed to be an infinite sequence of values; variables *buf* and *out* initially equal the empty sequence. The processes *Producer* and *Consumer* are given the identifiers 0 and 1. In PlusCal, an operation execution consists of execution of the code from one label to the next. Hence, the entire body of each process's **while** loop is executed atomically. The **await** statements assert enabling conditions of the actions. The keywords **fair** specify process fairness.

Figure 2 shows the first four states of an execution of the algorithm represented in the standard model. The letter P or C atop an arrow indicates which process's atomic step is executed to reach the next state.

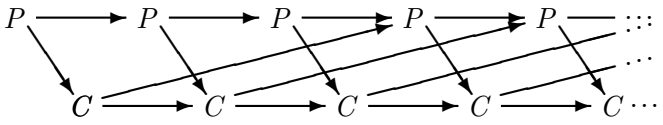


Figure 3: An event history for the FIFO queue with $N = 3$.

Algorithm PC is a specification; a bounded FIFO queue must implement that specification. A specification is a definition, and it makes no formal sense to ask if a definition is correct. However, we can gain confidence that this algorithm does specify a bounded FIFO queue by proving properties of it. The most important class of properties one proves about an algorithm are invariance properties. A state predicate is an *invariant* iff it is true in every state of every execution. The following invariant of algorithm PC suggests that it is a correct specification of an N-element bounded queue:

$$(Len(buf) \leq N) \ \wedge \ (Input = out \circ buf \circ in)$$

where $Len(buf)$ is the length of the sequence buf and $\circ$ is sequence concatenation.

The basic method for proving that a predicate Inv is an invariant of a concurrent algorithm was introduced by Edward Ashcroft in 1975 [2]. We find a suitable predicate I (the inductive invariant) and prove that (i) I is true in every initial state, (ii) I is left true by every step of the algorithm, and (iii) I implies Inv. It is easy to prove that the state predicate above is an invariant of algorithm PC. The appropriate inductive invariant I is the conjunction of this invariant with a predicate asserting that each variable has a "type-correct" value. (PlusCal is an untyped language.)

3.2 Another Way of Looking at a FIFO Queue

The FIFO queue specification allows only a single initial state, and executing either process's action can produce only a single next state. Hence the execution of Figure 2 is completely determined by the sequence $P \to C \to P \to P \to \cdots$ of atomic-action executions. For $N = 3$, all such sequences are described by the graph in Figure 3. The nodes of the graph are called *events*, each event representing an atomic execution of the algorithm step with which the event is labeled. The graph defines an irreflexive partial order $\prec$ on the events, where $e \prec f$ iff $e \neq f$ and there is a path through the graph from event e to event f. For want of a standard term for it, I will

call such a partially ordered set of events, in which events are labeled with atomic steps, an *event history*.

This event history describes all sequences of states that represent executions of algorithm PC in the standard model. Such a sequence of states is described by a sequence of infinitely many P and C events—that is, by a total ordering of the events in the event history. A total ordering of these events describes a possible execution of algorithm PC iff it is consistent with the partial order $\prec$. To see this, observe that the downward pointing diagonal arrows imply that the i^{th} P event (which moves the i^{th} input to the buffer) must precede the i^{th} C event (which moves that input from the buffer to the output). The upward pointing diagonal arrows indicate that the i^{th} C event must precede the $(i+3)^{\text{rd}}$ P event, which is necessary to ensure that there is room for the $(i+3)^{\text{rd}}$ input value in the buffer, which can hold at most 3 elements.

We can view the event history of the figure to be the single "real" execution of algorithm PC. The infinitely many different executions in the standard model are artifacts of the model; they are not inherently different. Two events not ordered by the $\prec$ relation—for example, the second C event and the fourth P event—represent operations that can be executed concurrently. However, the standard model requires concurrent executions of the two operations to be modeled as occurring in some order.

3.3 Mutual Exclusion versus Producer-Consumer Synchronization

Producer-consumer synchronization is inherently deterministic. On the other hand, mutual exclusion synchronization is inherently nondeterministic. It has an inherent race condition: two processes can compete to enter the critical section, and either might win.

Resolving a race requires an *arbiter*, a device that decides which of two events happens first [3]. An arbiter can take arbitrarily long to make its decision. (A well-designed arbiter has an infinitesimal probability of taking very long.) Any mutual exclusion algorithm can therefore, in principle, take arbitrarily long to allow some waiting process to enter its critical section. This is not an artifact of any model. It appears to be a law of nature.

Producer-consumer synchronization has no inherent nondeterminism, hence no race condition. It can be implemented without an arbiter, so each operation can be executed in bounded time. It is a fundamentally different class of problem than mutual exclusion.

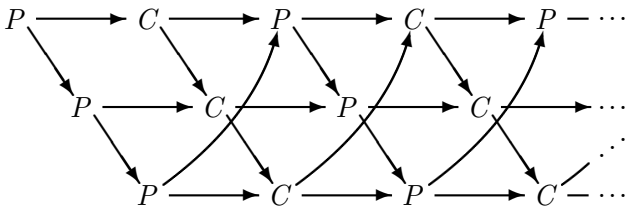


Figure 4: Another view of the FIFO queue for $N = 3$.

3.4 The FIFO Queue as an N-Process System

The graph in Figure 3 is drawn with two rows, each containing the events corresponding to actions of one of the two processes. Figure 4 is the same graph drawn with three rows. We can consider the three rows to be three separate processes. If we number these rows 0, 1, and 2 and we number the elements in the *Input* sequence starting from 0, then the events corresponding to the reading and outputting of element i of *Input* are in row i mod 3. We can consider each of those rows to be a process, making the FIFO queue a 3-process system for $N = 3$, and an N-process system in general. If we were to implement the variable *buf* with an N-element cyclic buffer, each of these processes would correspond to a separate buffer element.

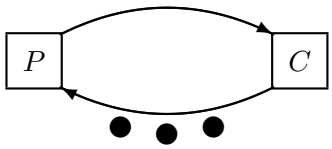
In the event history model, any totally ordered subset of events can be considered a process. The standard model has no inherent notion of processes. In that model, an execution is just a sequence of states. Processes are an artifact of the way the sequence of states is represented. The set of executions of algorithm *PC* can also be described by an N-process PlusCal algorithm.

3.5 Generalized Producer-Consumer Synchronization

The generalization of producer-consumer synchronization is marked-graph synchronization. Marked graphs were introduced by Holt and Commoner in 1970 [8]. A marked graph is a directed graph together with a *marking* that assigns a finite set of indistinguishable tokens to each arc. A node is *fired* in a marking by removing one token from each of its input arcs and adding one token to each of its output arcs (producing a new marking). A *firing sequence* of a marked graph is a sequence of firings that can end only with a marking in which no node may be fired. (By definition of firing, a node

can be fired iff it has at least one token on each input arc.)

A marked graph synchronization problem is described by labeling the nodes of a marked graph with the names of atomic operations. This specifies that a sequence of atomic operation executions is permitted iff it is the sequence of labels of the nodes in a possible firing sequence of the marked graph. For example, the following marked graph describes the FIFO queue for $N = 3$.



A token on the top arc represents a value in the buffer, and a token on the bottom arc represents space for one value in the buffer. Observe that the number of tokens on this marked graph remains constant throughout a firing sequence. The generalization of this observation to arbitrary marked graphs is that the number of tokens on any cycle remains constant.

All executions of a marked graph synchronization algorithm are described by a single event history. Marked graph synchronization can be implemented without an arbiter, so each operation can be executed in a bounded length of time.

Marked graphs can be viewed as a special class of Petri nets [18]. Petri nets are a model of concurrent computation especially well-suited for expressing the need for arbitration. Although simple and elegant, Petri nets are not expressive enough to formally describe most interesting concurrent algorithms. Petri nets have been used successfully to model some aspects of real systems, and they have been generalized to more expressive languages. But to my knowledge, neither Petri nets nor their generalizations have significantly influenced the field of concurrent algorithms.

3.6 The Two-Arrow Formalism Revisited

Let $\mathcal{E}$ be an event history with partial order $\prec$. Suppose we partition $\mathcal{E}$ into nonempty disjoint subsets called *operation executions*. We can define two relations $\rightarrow$ and $\dashrightarrow$ on the set of operation executions as follows, for any operation executions A and B:

$$A \rightarrow B \text{ iff } \forall\ e \in A, f \in B\ :\ e \prec f.$$

$$A \dashrightarrow B \text{ iff } \exists\ e \in A, f \in B\ :\ e \prec f.$$

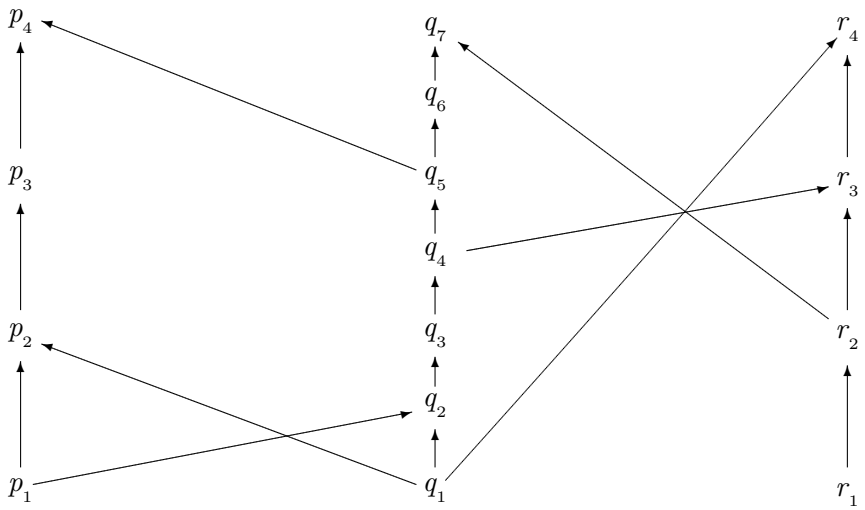


Figure 5: An Event History for a Distributed System

It is straightforward to see that these definitions (and the assumption that $\prec$ is an irreflexive partial order) imply properties $A1$–$A4$ of Section 2.3. Thus, we can obtain a two-arrow representation of the execution of an algorithm with non-atomic operations from an event history whose events are the atomic events that make up the operation executions. The event history does not have to be discrete. Its events could be points in a space-time continuum, where $\prec$ is the causality relation introduced by Minkowski [17].

4 Distributed Algorithms

Pictures of event histories were first used to describe distributed systems. Figure 5 is an event history that appeared as an illustration in [13]. The events come from three processes, with time moving upwards. A diagonal arc joining events from two different processes represents the causality relation requiring that a message must be sent before it is received. For example, the arc from q_4 to r_3 indicates that event q_4 of the second process sent a message that was received by event r_3 of the third process.

In general, executions of such a distributed system can produce different event histories. For example, in addition to the history of Figure 5, there might be an event history in which the message sent by event q_1 is received before the message sent by event q_4. In such a case, there is true nondeterminism and the system requires arbitration.

Let a *consistent* cut of an event system consist of a set C of events such

that for every two events c and d, if event c is in C and $d \prec c$, then d is in C. For example, $\{p_1, q_1, q_2, r_1, r_2\}$ is a consistent cut of the event history of Figure 5. Every consistent cut defines a global state of the system during some execution in the standard model—the state after executing the steps associated with the events in the consistent cut.

An event history like that of Figure 5 allows incompatible consistent cuts—that is two consistent cuts, neither of which is a subset of the other. They describe possible global states that, in the standard model, may not occur in the same execution. This shows that there is no meaningful concept of a unique global state at an instant. For example, there are different consistent cuts containing only events q_1 and q_2 of the second process. They represent different possible global states immediately after the process has executed event q_2. There is no reason to distinguish any of those global states as *the* global state at that instant.

Because the standard model refers to global states, it has been argued that the model should not be used for reasoning about distributed algorithms and systems. While this argument sounds plausible, it is wrong. An invariant of a global system is a meaningful concept because it is a state predicate that is true for all possible global states, and so does not depend on any preferred global states. The problem of implementing a distributed system can often be viewed as that of maintaining a global invariant even though different processes may have incompatible views of what the current state is at any instant.

Thinking is useful, and multiple ways of thinking can be even more useful. However, while event histories may be especially useful for helping us understand distributed systems, the best way to reason about these systems is usually in terms of global invariants. The standard model provides the most practical way to reason about invariance.

5 Afterwards

After distributed systems, the next major step in concurrent algorithms was the study of fault tolerance. The first scientific examination of fault tolerance was Dijkstra's seminal 1974 paper on self-stabilization [7]. However, as sometimes happens with work that is ahead of its time, that paper received little attention and was essentially forgotten for a decade. A survey of fault tolerance published in 1978 [20] does not mention a single algorithm, showing that fault tolerance was still the province of computer engineering, not of computer science.

At about the same time that the study of fault-tolerant algorithms began in earnest, the study of models of concurrency blossomed. Arguably, the most influential of this work was Milner's CCS [16]. These models were generally event-based, and avoided the use of state. They did not easily describe algorithms or the usual way of thinking about them based on the standard model. As a result, the study of concurrent algorithms and the study of formal models of concurrency split into two fields. A number of formalisms based on the standard model were introduced for describing and reasoning about concurrent algorithms. Notable among them is temporal logic, introduced by Amir Pnueli in 1977 [19].

The ensuing decades have seen a huge growth of interest in concurrency—particularly in distributed systems. Looking back at the origins of the field, what stands out is the fundamental role played by Edsger Dijkstra, to whom this history is dedicated.

References

[1] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

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

[3] J. C. Barros and B. W. Johnson. Equivalence of the arbiter, the synchronizer, the latch, and the inertial delay. *IEEE Transactions on Computers*, C-32(7):603–614, July 1983.

[4] Per Brinch Hansen. *Operating System Principles*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.

[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. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968. Originally appeared as EWD123 (1965).

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

[8] A. Holt and F. Commoner. Events and conditions. In *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, pages 3–52. Project MAC, June 1970.

[9] Harris Hyman. Comments on a problem in concurrent programming control. *Communications of the ACM*, 9(1):45, January 1966.

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

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

[12] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[13] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[14] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems*, 1(1):84–97, July 1979.

[15] Leslie Lamport. The PlusCal algorithm language. In Martin Leucker and Carroll Morgan, editors, *Theoretical Aspects of Computing, ICTAC 2009*, volume 5684 of *Lecture Notes in Computer Science*, pages 36–60. Springer-Verlag, 2009.

[16] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1980.

[17] H. Minkowski. Space and time. In *The Principle of Relativity*, pages 73–91. Dover, 1952.

[18] C. A. Petri. Fundamentals of a theory of asynchronous information flow. In Cicely M. Popplewell, editor, *Information Processing 1962, Proceedings of IFIP Congress 62*, pages 386–390. North-Holland, 1962.

[19] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, November 1977.

[20] B. Randell, P. A. Lee, and P.C. Treleaven. Reliability issues in computing system design. *Computing Surveys*, 10(2):123–165, June 1978.

[21] Brian A. Rudolph. Self-assessment procedure xxi. *Communications of the ACM*, 33(5):563–575, May 1990.