My Writings

Leslie Lamport

27 December 2011

This document contains descriptions of almost all my technical papers. Omitted are papers for which I no longer have copies and papers that are incomplete. I have also omitted early versions of some of these papers—even in cases where the title changed. Included are some initial drafts of papers that I abandoned before fixing errors or other problems in them. A table of contents precedes the descriptions. I also include a brief *curriculum vitae*.

Each description attempts to explain the genesis of the work. However, I have forgotten how I came to write most of my papers. Even when I discourse at length about the development of the ideas, I am giving only a subjective view based on my unreliable memory. Whenever possible, I have asked other people involved to check the accuracy of what I've written. However, what I have most often forgotten is the work of others that influenced my own work. This may give the impression that I am claiming more credit for ideas than I deserve, for which I apologize.

Where I think it's interesting, I give the story behind the publication or non-publication of a paper. Some of the stories read like complaints of unfair treatment by editors or referees. Such cases are bound to arise in any activity based on human judgment. On the whole, I have had little trouble getting my papers published. In fact, I have profited from the natural tendency of editors and referees to be less critical of the work of established scientists. But I think it's worth mentioning the cases where the system didn't work as it should.

I would like to have ordered my papers by the date they were written. However, I usually have no record of when I actually wrote something. So, I have ordered them by date of publication or, for unpublished works, by the date of the version that I have. Because of long and variable publication delays, this means that the order is only approximately chronological.

This is the printed version of the Web page currently at

`http://research.microsoft.com/users/lamport/pubs/pubs.`

`html`

That page can be found by searching the Web for the 23-letter string `alllamportspubsontheweb`. Please do not put this string in any document that might appear on the Web—including email messages and Postscript and Word documents. One way to refer to it in Web documents is:

> the 23-letter string obtained by removing the `-` characters from the string `alllam-portspu-bsonth-eweb`

Whenever possible, the web page includes electronic versions of the works. At the moment, I have electronic versions mainly of works written after about 1985. For journal articles, these may be "preprint" versions, formatted differently and sometimes differing slightly from the published versions. I hope in the future to provide scanned versions of earlier publications.

Education

- B.S. — M.I.T., 1960. (Mathematics)
- M.A. — Brandeis University, 1963. (Mathematics)
- Ph.D. — Brandeis University, 1972. (Mathematics)

Employment

- Mitre Corporation (part-time, 1962–1965)
- Marlboro College (1965–1969)
- Massachusetts Computer Associates (1970–1977)
- SRI International (1977–1985)
- Digital Equipment Corporation / Compaq (1985–2001)
- Microsoft Research (2001–present)

Honors

- National Academy of Engineering (1991)
- PODC Influential Paper Award (2000) (for paper [27])
- Honorary Doctorate, University of Rennes (2003)

- Honorary Doctorate, Christian Albrechts University, Kiel (2003)
- Honorary Doctorate, Ecole Polytechnique Fédérale de Lausanne (2004)
- IEEE Piore Award (2004)
- Edsger W. Dijkstra Prize in Distributed Computing (2005) (for paper [41])
- Honorary Doctorate, Universit della Svizzera Italiana, Lugano (2006)
- ACM SIGOPS Hall of Fame Award (2007) (for paper [27])
- Honorary Doctorate, Université Henri Poincaré, Nancy (2007)
- LICS 1988 Test of Time Award (2008) (for paper [92])
- IEEE John von Neumann Medal (2008)
- National Academy of Sciences (2011)

Contents

[1] **Braid Theory**. Mathematics Bulletin of the Bronx High School of Science (1957), pages 6,7, and 9.

This appears to be my first publication, written when I was a high school student. It shows that I was not a child prodigy.

[2] **Summer Vision Programs**. Massachusetts Institute of Technology, Project MAC Memorandum MAC-M-332, Artificial Intelligence Project Memo Number Vision 111 (October 1966).

In the summer of 1966, I worked at the M.I.T. Artificial Intelligence Laboratory, doing Lisp programming for a computer vision project. I have no memory of this document, but it appears to describe the programs I wrote that summer. It's of no technical interest, but it does show that, even in those days, I was writing precise documentation.

[3] **Preliminary User's Guide to Monitor 1** (with Roland Silver). Mitre Technical Report (December 1966).

While in graduate school, I worked summers and part-time at the Mitre Corporation from 1962 to 1965. I think I wrote three or four technical reports there, but this is the only one the Mitre library seems to have. A large part of my time at Mitre was spent working on the operating system for a computer being built there called Phoenix. This is the operating system's manual, apparently written by Silver based on work we had both done. There is nothing of technical interest here, but it provides a snapshot of what was going on in the world of computers in the early 60s.

[4] **Untitled Draft of Advanced Calculus Text**. Unpublished (*circa* 1967).

During the 1965–1969 academic years, I taught math at Marlboro College. I don't remember exactly when or how the project got started, but I wrote the first draft of an advanced calculus textbook for Prentice-Hall, from whom I received an advance of $500. (That sum, which seems ridiculously small now, was a significant fraction of my salary at the time.) The Prentice-Hall reviewers liked the draft. I remember one reviewer commenting that the chapter on exterior algebra gave him, for the first time, an intuitive understanding of the topic. However, because of a letter that was apparently lost in the mail, the Prentice-Hall editor and I both thought that the other had lost interest in the project. By the time this misunderstanding had been cleared up, I was ready to move on to other things and didn't feel like writing

a final draft. (This was before the days of computer text processing, so writing a new draft meant completely retyping hundreds of pages of manuscript.)

[5] **The Geometry of Space and Time**. Unpublished (*circa* 1968).

Marlboro College, where I taught math from 1965–1969, had a weekly series of lectures for the general public, each given by a faculty member or an outside speaker invited by a faculty member. I gave a lecture about relativity that I later turned into this short monograph. I made a half-hearted, unsuccessful effort to get it published. But it was too short (75 pages) to be a "real" book, and there was very little interest in science among the general public in the late sixties. I think this monograph is still a very good exposition of the subject. Unfortunately, the second half, on general relativity, is obsolete because it says nothing about black holes. While black holes appear in the earliest mathematical solutions to the equations of general relativity, it was only in the late 60s that many physicists began seriously to consider that they might exist and to study their properties.

[6] **Comment on Bell's Quadratic Quotient Algorithm**. *Communications of the ACM 13*, 9 (September 1970).

This short note describes a minor inefficiency I noticed in a hash-table algorithm published by James Bell. It got me thinking about hash tables, and I invented what I called the linear quotient algorithm—an algorithm that seems quite obvious in retrospect. While I was running simulations to gather data for a paper on that algorithm, the latest issue of *CACM* arrived with a paper by Bell and Charles Kaman titled *The Linear Quotient Hash Code*. I had devised three variants of the algorithm not contained in their article. So, I wrote a paper about those three variants and submitted it to *CACM*. The editor rejected it, without sending it out for review, saying that it was too small a contribution to merit publication. In the next few years, *CACM* published two papers by others on the subject, each completely devoted to one of my three variants. (Those papers had been submitted to different editors.) My paper, which was probably a Massachusetts Computer Associates (Compass) technical report, has been lost. (Compass went out of business a few years ago, and I presume that its library was destroyed.) The linear quotient method is probably the most common hash-coding algorithm used today.

[7] **The Analytic Cauchy Problem with Singular Data**. Ph.D. Thesis, Brandeis University (1972). Available from UMI, currently at `http://wwwlib.umi.com`, as number 7232105.

I left Marlboro College and went back to Brandeis in 1969 to complete my Ph.D. At that time, I intended to study and write a thesis in mathematical physics. However, I wound up doing a thesis in pure mathematics, on analytic partial differential equations. I learned nothing about analytic partial differential equations except what was needed for my thesis research, and I have never looked at them since then. The thesis itself was a small, solid piece of very classical math. Had Cauchy arisen from the grave to read it, he would have found nothing unfamiliar in the mathematics.

[8] **An Extension of a Theorem of Hamada on the Cauchy Problem with Singular Data**. *Bulletin of the Amer. Math. Society 79*, 4 (July 1973), 776–780.

At the time, and perhaps still today, a math student "copyrighted" his (seldom her) thesis results by announcing them in a short note in the *Bulletin of the AMS*. Normally, a complete paper would be published later. But I never did that, since I left math for computer science after completing my thesis.

[9] **The Coordinate Method for the Parallel Execution of DO Loops**. *Proceedings of the 1973 Sagamore Conference on Parallel Processing*, T. Feng, ed., 1–12.

Compass (Massachusetts Computer Associates) had a contract to write the Fortran compiler for the Illiac-IV computer, an array computer with 64 processors that all operated in lock-step on a single instruction stream. I developed the theory and associated algorithms for executing sequential DO loops in parallel on an array computer that were used by the compiler. The theory is pretty straightforward. The creativity lay in the proper mathematical formulation of the problem. Today, it would be considered a pretty elementary piece of work. But in those days, we were not as adept at applying math to programming problems. Indeed, when I wrote up a complete description of my work for my colleagues at Compass, they seemed to treat it as a sacred text, requiring spiritual enlightenment to interpret the occult mysteries of linear algebra.

Anyway, this paper was my first pass at writing up the complete version of the theory for publication. I strongly suspect that it has

never been read. No one seems to have noticed that, because of a text-editing error, the description of the algorithm is missing the punch line that says what can be executed in parallel. This paper is superseded by the unpublished [20].

[10] **The Parallel Execution of DO Loops**. *Communications of the ACM 17*, 2 (February 1974), 83–93.

This is the only journal paper to come out of the work mentioned in the description of [9]. It contains essentially the special case of the results in [9] for a single tight nesting of loops. It was one of the early articles on the topic, and I believe it was cited fairly often.

[11] **The Hyperplane Method for an Array Computer**. *Proceedings of the 1974 Sagamore Conference on Parallel Processing*, T. Feng, ed., Springer Verlag, 1–12.

In the late 19th century, the Vanderbilts (a rich American family) owned 500 hectares of beautiful land in the central Adirondack Mountains of New York State that included Sagamore Lake, which is about a kilometer in length. There, they built a rustic summer estate. In the 70s, and probably still today, the place was completely isolated, away from any other signs of civilization. The estate, with land and lake, was given to Syracuse University, which operated it as a conference center. An annual conference on parallel processing was held there late in the summers of 1973 through 1975. Sagamore was the most beautiful conference site I have ever seen. The conference wasn't bad, with a few good people attending, though it wasn't first rate. But I would have endured a conference on medieval theology for the opportunity to canoe on, swim in, and walk around the lake.

To justify my attendance at Sagamore, I always submitted a paper. But once I discovered that it was not a first-rate conference, I did not submit first-rate papers. However, I don't republish old material (except as necessary to avoid forcing people to read earlier papers), so this paper must have included some new results about the hyperplane method for parallelizing sequential loops. However, I can't find a copy of the paper and don't remember what was in it.

[12] **A New Solution of Dijkstra's Concurrent Programming Problem**. *Communications of the ACM 17*, 8 (August 1974), 453–455.

This paper describes the bakery algorithm for implementing mutual exclusion. I have invented many concurrent algorithms. I feel that I

did not invent the bakery algorithm, I discovered it. Like all shared-memory synchronization algorithms, the bakery algorithm requires that one process be able to read a word of memory while another process is writing it. (Each memory location is written by only one process, so concurrent writing never occurs.) Unlike any previous algorithm, and almost all subsequent algorithms, the bakery algorithm works regardless of what value is obtained by a read that overlaps a write. If the write changes the value from 0 to 1, a concurrent read could obtain the value 7456 (assuming that 7456 is a value that could be in the memory location). The algorithm still works. I didn't try to devise an algorithm with this property. I discovered that the bakery algorithm had this property after writing a proof of its correctness and noticing that the proof did not depend on what value is returned by a read that overlaps a write.

I don't know how many people realize how remarkable this algorithm is. Perhaps the person who realized it better than anyone is Anatol Holt, a former colleague at Massachusetts Computer Associates. When I showed him the algorithm and its proof and pointed out its amazing property, he was shocked. He refused to believe it could be true. He could find nothing wrong with my proof, but he was certain there must be a flaw. He left that night determined to find it. I don't know when he finally reconciled himself to the algorithm's correctness.

Several books have included emasculated versions of the algorithm in which reading and writing are atomic operations, and called those versions "the bakery algorithm". I find that deplorable. There's nothing wrong with publishing a simplified version, as long as it's called a simplified version.

What is significant about the bakery algorithm is that it implements mutual exclusion without relying on any lower-level mutual exclusion. Assuming that reads and writes of a memory location are atomic actions, as previous mutual exclusion algorithms had done, is tantamount to assuming mutually exclusive access to the location. So a mutual exclusion algorithm that assumes atomic reads and writes is assuming lower-level mutual exclusion. Such an algorithm cannot really be said to solve the mutual exclusion problem. Before the bakery algorithm, people believed that the mutual exclusion problem was unsolvable—that you could implement mutual exclusion only by using lower-level mutual exclusion. Brinch Hansen said exactly this in a 1972 paper. Many people apparently still believe it. (See [91].)

The paper itself does not state that it is a "true" mutual exclusion algorithm. This suggests that I didn't realize the full significance of the algorithm until later, but I don't remember.

For a couple of years after my discovery of the bakery algorithm, everything I learned about concurrency came from studying it. Papers like [25], [33], and [70] were direct results of that study. The bakery algorithm was also where I introduced the idea of variables belonging to a process—that is, variables that could be read by multiple processes, but written by only a single process. I was aware from the beginning that such algorithms had simple distributed implementations, where the variable resides at the owning process, and other processes read it by sending messages to the owner. Thus, the bakery algorithm marked the beginning of my study of distributed algorithms.

The paper contains one small but significant error. In a footnote, it claims that we can consider reads and writes of a single bit to be atomic. It argues that a read overlapping a write must get one of the two possible values; if it gets the old value, we can consider the read to have preceded the write, otherwise to have followed it. It was only later, with the work eventually described in [70], that I realized the fallacy in this reasoning.

[13] **On Self-stabilizing Systems**. Massachusetts Computer Associates Technical Report CA 7412-0511 (5 December 1974).

This note was written upon reading Dijkstra's classic paper "Self-stabilizing Systems in Spite of Distributed Control" that appeared in the November 1974 issue of *CACM* (see [58]). It generalizes one of the algorithms in Dijkstra's paper from a line of processes to an arbitrary tree of processes. It also discusses the self-stabilizing properties of the bakery algorithm. I never tried to publish this note—probably because I regarded it as too small a piece of work to be worth a paper by itself.

The note contains the intriguing sentence: "There is a complicated modified version of the bakery algorithm in which the values of all variables are bounded." I never wrote down that version, and I'm not sure what I had in mind. But I think I was thinking of roughly the following modification. As a process waits to enter its critical section, it keeps reducing its number, not entering the critical section until its number equals one. A process p can reduce its number by at most one, and only when the next lower-numbered process's number is at least two less than p's number, and the next higher-numbered process is within

one of p's number. I think I intended to use the techniques of [25] to allow reading and writing of numbers to remain non-atomic while maintaining the order of waiting processes. (If eventually all processes stop changing their numbers, then all processes will eventually read the correct numbers, allowing some process to progress.) At one time, I convinced myself that this algorithm is correct. But I never wrote a rigorous proof, so I don't know if it really works. Filling in the details and proving correctness should be a nice exercise.

[14] **On Programming Parallel Computers**. *Proceedings of a Conference on Programming Languages and Compilers for Parallel and Vector Machines*, published as *ACM SIGPLAN Notices 10*, 3 (March 1975), 25–33.

This is a position paper advocating the use of a higher-level language that expresses what must be computed rather than how it is to be computed. It argues that compilers are better than humans at generating efficient code for parallel machines. I was so naive then! I soon learned how bad compilers really were, and how trying to make them smarter just made them buggier. But compiler writers have gotten a lot better, so maybe this paper isn't as stupid now as it was then.

[15] **Parallel Execution on Array and Vector Computers**. *Proceedings of the 1975 Sagamore Conference on Parallel Processing*, T. Feng, ed., 187–191.

This paper considers the problem of efficiently executing a sequence of explicitly parallel statements—ones requiring simultaneous execution for all values of a parameter—when there are more parameter values than there are processors. It is one of the lesser papers that I saved for the Sagamore Conference. (See the discussion of [11].)

[16] **Multiple Byte Processing with Full-Word Instructions**. *Communications of the ACM 18*, 8 (August 1975), 471–475.

My algorithms for parallelizing loops, described in papers starting with [9], were rather inefficient. They could be sped up with parallel execution on an array processor like the Illiac-IV. But I realized one could do even better than the 64-times speedup provided by the Illiac's 64 processors. Each datum being manipulated was just a few bits, so I had the idea of packing several of the data into a single word and manipulating them simultaneously. Not only could this speed computation on the Illiac, but it allowed one to do array processing on

an ordinary uniprocessor. This paper describes general techniques for doing such parallel computation on packed data. It's a neat hack, and it's more useful now than it was then for two reasons. The obvious reason is that word size is larger now, with many computers having 64-bit words. The less obvious reason is that conditional operations are implemented with masking rather than branching. Instead of branching around the operation when the condition is not met, masks are constructed so the operation is performed only on those data items for which the condition is true. Branching is more costly on modern multi-issue computers than it was on the computers of the 70s.

[17] **The Synchronization of Independent Processes**. *Acta Informatica 7*, 1 (1976), 15–34.

There are a class of synchronization problems that are direct generalizations of mutual exclusion in that they assert constraints on when a process is allowed to perform a task. They include the dining philosophers problem and the readers/writers problem. This paper shows how a modified version of the bakery algorithm can be used to solve any such problem.

A referee said of the initial submission that the proofs were too long, tediously proving the obvious. In fact, there was a bug in the initial version, and at least one of those obvious proofs was of a false statement. Needless to say, I corrected the algorithm and wrote more careful proofs.

[18] **Comments on 'A Synchronization Anomaly'**. *Information Processing Letters 4*, 4 (January 1976), 88–89.

This is a comment on a short note by Richard Lipton and Robert Tuttle claiming to find an inadequacy in Dijkstra's P and V synchronization primitives. It points out that they had introduced a red herring because the problem that those primitives couldn't solve could not be stated in terms of entities observable within the system. As my note states: "A system cannot be correct unless its correctness depends only upon events and conditions observable within the system." That's something worth remembering, since I've encountered that same sort of red herring on other occasions. My observation is relevant to [63], but I had forgotten all about this note by the time I wrote [63].

[19] **Garbage Collection with Multiple Processes: an Exercise in**

Parallelism. *Proceedings of the 1976 International Conference on Parallel Processing*, T. Feng, ed., 50–54.

This is a minor extension to the concurrent garbage collection algorithm of [31]. That algorithm uses a single garbage collector process running in parallel with the "mutator" process that creates the garbage. This paper describes how to use multiple processes to do the collection, and how to handle multiple mutator processes. It is a minor work that I wrote up as an excuse for going to the Sagamore conference. (See the discussion of [11].) However, the conference had been renamed and moved from Sagamore to a less attractive, mosquito-infected site in the Michigan woods. It was the last time I attended.

[20] **The Coordinate Method for the Parallel Execution of Iterative Loops**. Unpublished (August 1976).

This is a totally revised version of [9], complete with proofs (which had been omitted from [9]). I submitted it to *CACM*, but the editor of *CACM* decided that it was more appropriate for *JACM*. When I submitted it there, the editor of *JACM* rejected it without sending it out for review because the basic ideas had already appeared in the conference version [9]. I was young and inexperienced, so I just accepted the editor's decision. I began to suspect that something was amiss a year or two later, when a paper from the same conference was republished verbatim in *CACM*. By the time I realized how crazy the editor's decision had been, it didn't seem worth the effort of resubmitting the paper.

[21] **Towards a Theory of Correctness for Multi-User Data Base Systems**. Rejected by the *1977 IFIP Congress* (October 1976).

This paper was a rough draft of some ideas, not all of which were correct. I don't remember how it became known, but I received requests for copies for years afterwards.

[22] **On the Glitch Phenomenon** (with Richard Palais). Rejected by *IEEE Transactions on Computers* (November 1976).

When I wrote [12], a colleague at Massachusetts Computer Associates pointed out that the concurrent reading and writing of a single register, assumed in the bakery algorithm, requires an arbiter—a device for making a binary decision based on inputs that may be changing. In the early 70s, computer designers rediscovered that it's impossible to build an arbiter that is guaranteed to reach a decision in a bounded length of

time. (This had been realized in the 50s but had been forgotten.) My colleague's observation led to my interest in the arbiter problem—or "glitch" problem, as it was sometimes called.

The basic proof that an arbiter cannot have a bounded response time uses continuity to demonstrate that, if there are two inputs that can drive a flip-flop into two different states, then there must exist an input that makes the flip-flop hang. At the time, it was very difficult to convince someone that this argument was valid. They seemed to believe that, because a flip-flop has only discrete stable states, continuity doesn't apply.

I described the arbiter problem to Palais, who had been my *de jure* thesis adviser and afterwards became a colleague and a friend. He recognized that the correct mathematical way to view what was going on is in terms of the compact-open topology on the space of flip-flop behaviors. So, we wrote this paper to explain why the apparently discontinuous behavior of an arbiter is actually continuous in the appropriate topology.

This paper was rejected by the *IEEE Transactions on Computers* because the engineers who reviewed it couldn't understand the mathematics. Six years later, the journal apparently acquired more mathematically sophisticated reviewers, and it published a less general result with a more complicated proof. I believe someone has finally published a paper on the subject that does supersede ours.

[23] **Proving the Correctness of Multiprocess Programs**. *IEEE Transactions on Software Engineering SE-3*, 2 (March 1977), 125–143.

When I first learned about the mutual exclusion problem, it seemed easy and the published algorithms seemed needlessly complicated. So, I dashed off a simple algorithm and submitted it to *CACM*. I soon received a referee's report pointing out the error. This had two effects. First, it made me mad enough at myself to sit down and come up with a real solution. The result was the bakery algorithm described in [12]. The second effect was to arouse my interest in verifying concurrent algorithms. This has been a very practical interest. I want to verify the algorithms that I write. A method that I don't think is practical for my everyday use doesn't interest me.

In the course of my work on parallelizing sequential code (see [10]), I essentially rediscovered Floyd's method as a way of extracting properties of a program. When I showed a colleague what I was doing, he went to our library at Massachusetts Computer Associates and gave

me a copy of the original tech report version of Floyd's classic paper *Assigning Meanings to Programs*. I don't remember when I read Hoare's *An Axiomatic Basis for Computer Programming*, but it was probably not long afterwards.

In the mid-70s, several people were thinking about the problem of verifying concurrent programs. The seminal paper was Ed Ashcroft's *Proving Assertions About Parallel Programs*, published in the Journal of Computer and System Sciences in 1975. That paper introduced the fundamental idea of invariance. I discovered how to use the idea of invariance to generalize Floyd's method to multiprocess programs. As is so often the case, in retrospect the idea seems completely obvious. However, it took me a while to come to it. I remember that, at one point, I thought that a proof would require induction on the number of processes.

This paper introduced the concepts of safety and liveness as the proper generalizations of partial correctness and termination to concurrent programs. It also introduced the terms "safety" and "liveness" for those classes of properties. I stole those terms from Petri nets, where they have similar but formally very different meanings. (Safety of a Petri net is a particular safety property; liveness of a Petri net is not a liveness property.)

At the same time I was devising my method, Susan Owicki was writing her thesis at Cornell under David Gries and coming up with very much the same ideas. The generalization of Floyd's method for proving safety properties of concurrent programs became known as the Owicki-Gries method. Owicki and Gries did not do anything comparable to the method for proving liveness in my paper. (Nissim Francez and Amir Pnueli developed a general proof method that did handle liveness properties, but it lacked a nice way of proving invariance properties.) My method had deficiencies that were corrected with the introduction of temporal logic, discussed in [47].

The Owicki-Gries version of the method for proving safety properties differed from mine in two ways. The significant way was that I made the control state explicit, while they had no way to talk about it directly. Instead, they introduced dummy variables to capture the control state. The insignificant way was that I used a flowchart language while they used an Algol-like language.

The insignificant syntactic difference in the methods turned out to have important ramifications. For writing simple concurrent algorithms, flowcharts are actually better than conventional toy program-

ming languages because they make the atomic actions, and hence the control state, explicit. However, by the mid-70s, flowcharts were passé and structured programming was all the rage, so my paper was forgotten and people read only theirs. The paper was rediscovered about ten years later, and is now generally cited alongside theirs in the mandatory references to previous work.

More important though is that, because they had used a "structured" language, Owicki and Gries thought that they had generalized Hoare's method. To the extent that Floyd's and Hoare's methods are different, it is because Hoare's method is based on the idea of hierarchical decomposition of proofs. The Owicki-Gries method doesn't permit this kind of clean hierarchical decomposition. Gries, commenting in 1999, said: "We hardly ever looked at Floyd's work and simply did everything based on Hoare's axiomatic theory." I suspect that, because they weren't thinking at all about Floyd's approach, they didn't notice the difference between the two, and thus they didn't realize that they were generalizing Floyd's method and not Hoare's.

The result of presenting a generalization of Floyd's method in Hoare's clothing was to confuse everyone. For a period of about ten years, hardly anyone really understood that the Owicki-Gries method was just a particular way of structuring the proof of a global invariant. I can think of no better illustration of this confusion than the EWD[1] *A Personal Summary of the Owicki-Gries Theory* that Dijkstra wrote and subsequently published in a book of his favorite EWDs. If even someone as smart and generally clear-thinking as Dijkstra could write such a confusing hodge-podge of an explanation, imagine how befuddled others must have been. A true generalization of Hoare's method to concurrent programs didn't come until several years later in [40].

I think it soon became evident that one wanted to talk explicitly about the control state. Susan Owicki obviously agreed, since we introduced the *at*, *in*, and *after* predicates for doing just that in [47]. Quite a bit later, I had more to say about dummy variables versus control state in [78].

Dummy variables were more than just an ugly hack to avoid control variables. They also allowed you to capture history. Adding history variables makes it possible to introduce behavioral reasoning into an assertional proof. (In the limit, you can add a variable that captures

[1] Throughout his career, Edsger Dijkstra wrote a series of notes identified by an "EWD" number.

the entire history and clothe a completely behavioral proof in an assertional framework.) What a program does next depends on its current state, not on its history. Therefore, a proof that is based on a history variable doesn't capture the real reason why a program works. I've always found that proofs that don't use history variables teach you more about the algorithm. (As shown in [92], history variables may be necessary if the correctness conditions themselves are in terms of history.)

When we developed our methods, Owicki and I and most everyone else thought that the Owicki-Gries method was a great improvement over Ashcroft's method because it used the program text to decompose the proof. I've since come to realize that this was a mistake. It's better to write a global invariant. Writing the invariant as an annotation allows you to hide some of the explicit dependence of the invariant on the control state. However, even if you're writing your algorithm as a program, more often than not, writing the invariant as an annotation rather than a single global invariant makes things more complicated. But even worse, an annotation gets you thinking in terms of separate assertions rather than in terms of a single global invariant. And it's the global invariant that's important. Ashcroft got it right. Owicki and Gries and I just messed things up. It took me quite a while to figure this out.

Sometime during the '80s, Jay Misra noticed that the definition of well-foundedness (Definition 8 on page 136) is obviously incorrect.

[24] **Formal Correctness Proofs for Multiprocess Algorithms**. *Programmation–2*$^{\text{me}}$ *Colloque International*, B. Robinet, ed., Dunod, Paris (1977), 1–8.

This is an abbreviated, conference version of [23]. (In those days, publication was fast enough that the journal version could appear before the conference version.) I wrote this paper as an excuse for attending a conference in Paris. However, it turned out that I went to Europe for other reasons that made it impossible for me to attend the conference. I tried to withdraw the paper, but it was too late.

[25] **On Concurrent Reading and Writing**. *Communications of the ACM 20*, 11 (November 1977), 806–811.

This paper came out of my study of the bakery algorithm of [12]. The problem with that algorithm is that it requires unbounded state. To allow the state to be bounded in practice, I needed an algorithm for

reading and writing multidigit numbers, one digit at a time, so that a read does not obtain too large a value if it overlaps a write. This paper shows that this can be done by simply writing the digits in one direction and reading them in the other. It also has some other nice algorithms.

The paper assumes that reading and writing a single digit are atomic operations. The original version introduced the notion of a regular register and proved the results under the weaker assumption that the individual digits were regular. However, the editor found the idea of nonatomic reads and writes to individual digits too heretical for *CACM* readers, and he insisted that I make the stronger assumption of atomicity. So, the world had to wait another decade, until the publication of [70], to learn about regular registers.

[26] **State the Problem Before Describing the Solution**. *ACM SIGSOFT Software Engineering Notes 3*, 1 (January 1978) 26.

The title says it all. This one-page note is as relevant today as when I wrote it. Replace "describing the solution" by "writing the program" and it becomes a practical recipe for improving software.

[27] **Time, Clocks and the Ordering of Events in a Distributed System**. *Communications of the ACM 21*, 7 (July 1978), 558–565. Reprinted in several collections, including *Distributed Computing: Concepts and Implementations*, McEntire et al., ed. IEEE Press, 1984.

Jim Gray once told me that he had heard two different opinions of this paper: that it's trivial and that it's brilliant. I can't argue with the former, and I am disinclined to argue with the latter.

The origin of this paper was a note titled *The Maintenance of Duplicate Databases* by Paul Johnson and Bob Thomas. I believe their note introduced the idea of using message timestamps in a distributed algorithm. I happen to have a solid, visceral understanding of special relativity (see [5]). This enabled me to grasp immediately the essence of what they were trying to do. Special relativity teaches us that there is no invariant total ordering of events in space-time; different observers can disagree about which of two events happened first. There is only a partial order in which an event e_1 precedes an event e_2 iff e_1 can causally affect e_2. I realized that the essence of Johnson and Thomas's algorithm was the use of timestamps to provide a total ordering of events that was consistent with the causal order. This realization may have been brilliant. Having realized it,

everything else was trivial. Because Thomas and Johnson didn't understand exactly what they were doing, they didn't get the algorithm quite right; their algorithm permitted anomalous behavior that essentially violated causality. I quickly wrote a short note pointing this out and correcting the algorithm.

It didn't take me long to realize that an algorithm for totally ordering events could be used to implement any distributed system. A distributed system can be described as a particular sequential state machine that is implemented with a network of processors. The ability to totally order the input requests leads immediately to an algorithm to implement an arbitrary state machine by a network of processors, and hence to implement any distributed system. So, I wrote this paper, which is about how to implement an arbitrary distributed state machine. As an illustration, I used the simplest example of a distributed system I could think of—a distributed mutual exclusion algorithm.

This is my most often cited paper. Many computer scientists claim to have read it. But I have rarely encountered anyone who was aware that the paper said anything about state machines. People seem to think that it is about either the causality relation on events in a distributed system, or the distributed mutual exclusion problem. People have insisted that there is nothing about state machines in the paper. I've even had to go back and reread it to convince myself that I really did remember what I had written.

The paper describes the synchronization of logical clocks. As something of an afterthought, I decided to see what kind of synchronization it provided for real-time clocks. So, I included a theorem about real-time synchronization. I was rather surprised by how difficult the proof turned out to be. This was an indication of what lay ahead in [62].

This paper won the 2000 PODC Influential Paper Award (later renamed the Edsger W. Dijkstra Prize in Distributed Computing). It won an ACM SIGOPS Hall of Fame Award in 2007.

[28] **The Specification and Proof of Correctness of Interactive Programs**. *Proceedings of the International Conference on Mathematical Studies of Information Processing* Kyoto, Japan (August, 1978), 477–540.

In the late 70s, people were talking about designing programming languages that would make program verification easier. I didn't think much of that idea. I felt that the difficulty in verification comes from the algorithm, not the details of the programming language in which

it is described. To demonstrate this view, I published in this paper a proof of correctness of a TECO program.

TECO stands for Text Editing and Correction. It was the command language for the TECO editor, and it was the underlying macro language on which the original version of Emacs was built. It was an obscure, low-level language whose goal was to perform powerful text editing operations with the minimum number of keystrokes. A programming language designed to make verification easy would be completely unlike TECO. The paper shows that you verify a TECO program the same way you verify a program written in a more conventional language.

The proof is perhaps also of some historical interest because it was an early example of a proof of an interactive program—that is, one that interacts with the user instead of just producing an answer. Thus, correctness had to be asserted in terms of the sequence of input actions. The paper generalizes the Floyd/Hoare method to deal with the history of environment actions.

[29] **The Implementation of Reliable Distributed Multiprocess Systems**. *Computer Networks 2* (1978), 95–114.

In [27], I introduced the idea of implementing any distributed system by using an algorithm to implement an arbitrary state machine in a distributed system. However, the algorithm in [27] assumed that processors never fail and all messages are delivered. This paper gives a fault-tolerant algorithm. It's a real-time algorithm, assuming upper bounds on message delays in the absence of faults, and that nonfaulty processes had clocks synchronized to within a known bound.

To my knowledge, this is the first published paper to discuss arbitrary failures (later called Byzantine failures). It actually considered malicious behavior, not using such behavior simply as a metaphor for completely unpredictable failures. Its algorithm was the inspiration for the digital signature algorithm of [41]. With its use of real-time, this paper presaged the ideas in [55].

[30] **SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control** (with John Wensley et al.). *Proceedings of the IEEE 66*, 10 (October 1978), 1240–1255.

When it became clear that computers were going to be flying commercial aircraft, NASA began funding research to figure out how to make them reliable enough for the task. Part of that effort was the SIFT

project at SRI. This project was perhaps most notable for producing the Byzantine generals problem and its solutions, first reported in [41].

This paper gives an overview of the complete SIFT project, which included designing the hardware and software and formally verifying the system's correctness. It announces the results that appear in [41]. It also is a very early example of the basic specification and verification method I still advocate: writing a specification as a state-transition system and showing that each step of the lower-level specification either implements a step of the higher-level one or is a "stuttering" step that leaves the higher-level state unchanged. The paper doesn't mention the use of an invariant, but that was probably omitted to save space.

This paper was a group effort that I choreographed in a final frenzy of activity to get the paper out in time for the issue's deadline. I don't remember who wrote what, but the section on verification seems to be my writing.

[31] **On-the-fly Garbage Collection: an Exercise in Cooperation** (with Edsger Dijkstra et al.). *Communications of the ACM 21*, 11 (November 1978), 966–975.

This paper presents the first concurrent garbage collection algorithm—that is, an algorithm in which the collector operates concurrently with the process that creates the garbage. The paper is fairly well known; its history is not.

I received an early version of the paper from Dijkstra, and I made a couple of suggestions. Dijkstra incorporated them and, quite generously, added me to the list of authors. He submitted the paper to *CACM*. The next I heard about it was when I received a copy of a letter from Dijkstra to the editor withdrawing the paper. The letter said that someone had found an error in the algorithm, but gave no indication of what the error was. Since Dijkstra's proof was so convincing, I figured that it must be a trivial error that could easily be corrected.

I had fairly recently written [23]. So, I decided to write a proof using that proof method, thinking that I would then find and correct the error. In about 15 minutes, trying to write the proof led me to the error. To my surprise, it was a serious error.

I had a hunch that the algorithm could be fixed by changing the order in which two operations were performed. But I had no good reason to believe that would work. Indeed, I could see no simple

informal argument to show that it worked. However, I decided to go ahead and try to write a formal correctness proof anyway. It took me about two days of solid work, but I constructed the proof. When I was through, I was convinced that the algorithm was now correct, but I had no intuitive understanding of why it worked.

In the meantime, Dijkstra figured out that the algorithm could be fixed by interchanging two other operations, and he wrote the same kind of behavioral proof as before. His fix produced an arguably more efficient algorithm than mine, so that's the version we used. I sketched an assertional proof of that algorithm. Given the evidence of the unreliability of his style of proof, I tried to get Dijkstra to agree to a rigorous assertional proof. He was unwilling to do that, though he did agree to make his proof somewhat more assertional and less operational. Here are his comments on that, written in July, 2000:

> 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 (CACM, Sep. 1965) with the mutual exclusion.

I, too, have always been most interested in showing that something could be done, rather than in finding a better algorithm for doing what was known to be possible. Perhaps that is why I have always been so impressed by the brilliance of Dijkstra's work on concurrent algorithms.

David Gries later published an Owicki-Gries style proof of the algorithm that was essentially the same as the one I had sketched. He simplified things a bit by combining two atomic operations into one. He mentioned that in a footnote, but *CACM* failed to print the footnote. (However, they did print the footnote number in the text.)

The lesson I learned from this is that behavioral proofs are unre-

liable and one should always use state-based reasoning for concurrent algorithms—that is, reasoning based on invariance.

[32] **A General Construction for Expressing Repetition**. *ACM SIGPLAN Notices 14*, 3 (March 1979) 38–42.

Fortunately, this note has been completely forgotten. It was written when people were still proposing new programming-language constructs. The one presented here may have little to recommend it, but it was no worse than many others.

[33] **A New Approach to Proving the Correctness of Multiprocess Programs**. *ACM Transactions on Programming Languages and Systems 1*, 1 (July 1979), 84–97.

Like everyone else at the time, when I began studying concurrent algorithms, I reasoned about them behaviorally. Such reasoning typically involved arguments based on the order in which events occur. I discovered that proofs can be made simpler, more elegant, and more mathematical by reasoning about operations (which can be composed of multiple events) and two relations on them: *precedes* (denoted by a solid arrow) and *can affect* (denoted by a dashed arrow). Operation A precedes operation B if all the events of A precede all the events of B; and A can affect B if some event in A precedes some event in B. These relations obey some simple rules that can reduce behavioral reasoning to mathematical symbol pushing.

This paper introduced the method of reasoning with the two arrow relations and applied it to a variant of the bakery algorithm. In the spring of 1976 I spent a month working with Carel Scholten at the Philips *Nat Lab* in Eindhoven. Scholten was a colleague and friend of Dijkstra, and the three of us spent one afternoon a week working, talking, and drinking beer at Dijkstra's house. The algorithm emerged from one of those afternoons. I think I was its primary author, but as I mention in the paper, the beer and the passage of time made it impossible for me to be sure of who was responsible for what.

The solid and dashed arrow formalism provides very elegant proofs for tiny algorithms such as the bakery algorithm. But, like all behavioral reasoning, it is hard to make completely formal, and it collapses under the weight of a complex problem. You should use assertional methods to reason about complex algorithms. However, standard assertional reasoning requires that the algorithm be written in terms of its atomic operations. The only assertional approach to reasoning

directly about nonatomic operations (without translating them into sequences of atomic operations) is the one in [86], which is not easy to use. The two-arrow formalism is still good for a small class of problems.

The formalism seems to have been almost completely ignored, even among the theoretical concurrency community. I find this ironic. There is a field of research known by the pretentious name of "true concurrency". Its devotees eschew assertional methods that are based on interleaving models because such models are not truly concurrent. Instead, they favor formalisms based on modeling a system as a partial ordering of events, which they feel is the only truly concurrent kind of model. Yet, those formalisms assume that events are atomic and indivisible. Atomic events don't overlap in time the way real concurrent operations do. The two-arrow formalism is the only one I know that is based on nonatomic operations and could therefore be considered truly concurrent. But, as far as I know, the true concurrency community has paid no attention to it.

[34] **How to Present a Paper**. Unpublished note, 4 August 1979.

This three-page note is about presenting a paper at a conference, but it offers good advice for any talk. Except for a couple of suggestions about hand-written slides, it still applies today.

[35] **How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs**. *IEEE Transactions on Computers C-28*, 9 (September 1979) 690–691.

I forget what prompted me to be thinking about memory caching, but it occurred to me one day that multiprocessor synchronization algorithms assume that each processor accesses the same word in memory, but each processor actually accesses its own copy in its cache. It hardly required a triple-digit IQ to realize that this could cause problems. I suppose what made this paper worth reading was its simple, precise definition of sequential consistency as the required correctness condition. This was not the first paper about cache coherence. However, it is the early paper most often cited in the cache-coherence literature.

[36] **Constructing Digital Signatures from a One Way Function**. SRI International Technical Report CSL–98 (October 1979).

At a coffee house in Berkeley around 1975, Whitfield Diffie described a problem to me that he had been trying to solve: constructing a

digital signature for a document. I immediately proposed a solution. Though not very practical—it required perhaps 64 bits of published key to sign a single bit—it was the first digital signature algorithm. Diffie and Hellman mention it in their classic paper:

> Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory IT-22*, 6 (1976), 644-654.

(I think it's at the bottom right of page 650.)

In 1978, Michael Rabin published a paper titled *Digitalized Signatures* containing a more practical scheme for generating digital signatures of documents. (I don't remember what other digital signature algorithms had already been proposed.) However, his solution had some drawbacks that limited its utility. This report describes an improvement to Rabin's algorithm that eliminates those drawbacks.

I'm not sure why I never published this report. However, I think it was because, after writing it, I realized that the algorithm could be fairly easily derived directly from Rabin's algorithm. So, I didn't feel that it added much to what Rabin had done. However, I've been told that this paper is cited in the cryptography literature and is considered significant, so perhaps I was wrong.

[37] **On the Proof of Correctness of a Calendar Program**. *Communications of the ACM 22*, 10 (October 1979), 554–556.

In the May, 1978 *CACM*, Matthew Geller published a paper titled *Test Data as an Aid in Proving Program Correctness*. He argued that there were some programs whose correctness is so hard to state formally that formally verifying them is useless because the specification is likely to be wrong. He gave as an example a program to compute the number of days between two dates in the same year, claiming that it would be hard to check the correctness of a precise statement of what it meant for the program to be correct. This paper proved him wrong. (It also makes the amusing observation that Geller's solution is wrong because it fails for dates before the advent of the Gregorian calendar.) As a bonus, readers of this paper were alerted well in advance that the year 2000 is a leap year.

[38] **Letter to the Editor**. *Communications of the ACM 22*, 11 (November 1979), 624.

In the May, 1979 *CACM*, De Millo, Lipton, and Perlis published an

influential paper titled *Social Process and Proofs of Theorems and Programs*. This paper made some excellent observations. However, by throwing in a few red herrings, they came to some wrong conclusions about program verification. More insidiously, they framed the debate as one between a reasonable engineering approach that completely ignores verification and a completely unrealistic view of verification advocated only by its most naive proponents. (There were, unfortunately, quite a few such proponents.)

At that time, some ACM publications had a special section on algorithms. In an ironic coincidence, the same issue of *CACM* carried the official ACM policy on algorithm submissions. It included all sorts of requirements on the form of the code, and even on the comments. Missing was any requirement that the correctness of the algorithm be demonstrated in any way.

I was appalled at the idea that, ten years after Floyd and Hoare's work on verification, the ACM was willing to publish algorithms with no correctness argument. The purpose of my letter was to express my dismay. I ironically suggested that they had succumbed to the arguments of De Millo, Lipton, and Perlis in their policy. As a result, my letter was published as a rebuttal to the De Millo, Lipton, and Perlis paper. No one seems to have taken it for what it was—a plea to alter the ACM algorithms policy to require that there be some argument to indicate that an algorithm worked.

[39] **'Sometime' is Sometimes 'Not Never'**. *Proceedings of the Seventh ACM Symposium on Principles of Programming Languages*, ACM SIGACT-SIGPLAN (January 1980).

After graduating from Cornell, Susan Owicki joined the faculty of Stanford. Some time around 1978, she organized a seminar to study the temporal logic that Amir Pnueli had recently introduced to computer science. I was sure that temporal logic was some kind of abstract nonsense that would never have any practical application, but it seemed like fun, so I attended. I observed that people got very confused because, in Pnueli's logic, the concepts of *always* and *eventually* mean what they do to ordinary people. In particular, something is not always true if and only if it is eventually false. (It doesn't always rain means that it eventually stops raining.) However, most computer scientists have a different way of thinking. For them, something is not always true if and only if it might possibly become false. (The program doesn't always produce the right answer means that it might

produce the wrong answer.)

I realized that there are two types of temporal logic: the one Pnueli used I called *linear time* logic; the one most computer scientists seemed to find natural I called *branching time*. (These terms were used by temporal logicians, but they distinguished the two logics by the axioms they satisfied, while I described them in terms of different kinds of semantics.) Pnueli chose the right kind of logic—that is, the one that is most useful for expressing properties of concurrent systems. I wrote this paper to explain the two kinds of logic, and to advocate the use of linear-time logic. However, I figured that the paper wouldn't be publishable without some real theorems. So, I proved some simple results demonstrating that the two kinds of logic really are different.

I submitted the paper to the Evian Conference, a conference on concurrency held in France to which it seems that everyone working in the field went. I was told that my paper was rejected because they accepted a paper by Pnueli on temporal logic, and they didn't feel that such an obscure subject merited two papers. I then submitted the paper to FOCS, where it was also rejected. I have very rarely resubmitted a paper that has been rejected. Fortunately, I felt that this paper should be published. It has become one of the most frequently cited papers in the temporal-logic literature.

[40] **The 'Hoare Logic' of Concurrent Programs**. *Acta Informatica 14*, 1 (1980), 21–37.

As explained in the discussion of [23], the Owicki-Gries method and its variants are generalizations of Floyd's method for reasoning about sequential programs. Hoare's method formalizes Floyd's with a set of axioms for deriving triples of the form $\{P\}S\{Q\}$, which means that if statement S is executed from a state in which P is true and terminates, then Q will be true. This paper introduces the generalization of Hoare's method to concurrent programs, replacing Hoare triples with assertions of the form $\{I\}S$, which means that the individual actions of statement S leave the predicate I invariant.

When I wrote this paper, I sent a copy to Tony Hoare thinking that he would like it. He answered with a letter that said, approximately: "I always thought that the generalization to concurrent programs would have to look something like that; that's why I never did it." (Unfortunately, I no longer have the letter.) At the time, I dismissed his remark as the ramblings of an old fogey. I now think he was right—though probably for different reasons than he does. As I indicated in the dis-

cussion of [23], I think Ashcroft was right; one should simply reason about a single global invariant, and not do this kind of decomposition based on program structure.

[41] **Reaching Agreement in the Presence of Faults** (with Marshall Pease and Robert Shostak). *Journal of the Association for Computing Machinery 27*, 2 (April 1980).

Before this paper, it was generally assumed that a three-processor system could tolerate one faulty processor. This paper shows that "Byzantine" faults, in which a faulty processor sends inconsistent information to the other processors, can defeat any traditional three-processor algorithm. (The term *Byzantine* didn't appear until [46].) In general, $3n+1$ processors are needed to tolerate n faults. However, if digital signatures are used, $2n+1$ processors are enough. This paper introduced the problem of handling Byzantine faults. I think it also contains the first precise statement of the consensus problem.

I am often unfairly credited with inventing the Byzantine agreement problem. The problem was formulated by people working on SIFT (see [30]) before I arrived at SRI. I had already discovered the problem of Byzantine faults and written [29]. (I don't know if that was earlier than or concurrent with its discovery at SRI.) However, the people at SRI had a much simpler and more elegant statement of the problem than was present in [29].

The 4-processor solution presented in this paper and the general impossibility result were obtained by Shostak; Pease invented the $3n+1$-processor solution. My contribution to the work in this paper was the solution using digital signatures, which is based on the algorithm in [29]. It was my work on digital signatures (see [36]) that led me to think in that direction. However, digital signatures, as used here, are a metaphor. Since the signatures need be secure only against random failure, not against an intelligent adversary, they are much easier to implement than true digital signatures. However, this point seems to have escaped most people, so they rule out the algorithms that use digital signatures because true digital signature algorithms are expensive. Thus, $3n+1$-processor solutions are used even though there are satisfactory $2n+1$-processor solutions,

My other contribution to this paper was getting it written. Writing is hard work, and without the threat of perishing, researchers outside academia generally do less publishing than their colleagues at universities. I wrote an initial draft, which displeased Shostak so much that

he completely rewrote it to produce the final version.

Over the years, I often wondered whether the people who actually build airplanes know about the problem of Byzantine failures. In 1997, I received email from John Morgan who used to work at Boeing. He told me that he came across our work in 1986 and that, as a result, the people who build the passenger planes at Boeing are aware of the problem and design their systems accordingly. But, in the late 80s and early 90s, the people at Boeing working on military aircraft and on the space station, and the people at McDonnell-Douglas, did not understand the problem. I have no idea what Airbus knows or when they knew it.

This paper was awarded the 2005 Edsger W. Dijkstra Prize in Distributed Computing.

[42] **Program Verification: An Approach to Reliable Hardware and Software** (with J S. Moore). *Transactions of the American Nuclear Society 35* (November 1980), 252–253.

In 1980, J Moore and I were both at SRI and had been involved in the verification of SIFT (see [30]). The nuclear power industry was, for obvious reasons, interested in the correctness of computer systems. I forget how it came to pass, but Moore and I were invited to present a paper on verification at some meeting of power industry engineers. This was the result.

[43] **Password Authentication with Insecure Communication**. *Communications of the ACM 24*, 11 (November 1981), 770–772.

Despite a casual interest in civilian cryptography going back to its origins (see the discussion of [36]), this is my only publication in the field. It presents a cute hack for using a single password to login to a system multiple times without allowing an adversary to gain access to the system by eavesdropping. This hack is the basis of Bellcore's S/KEY system and of the PayWord system of Rivest and Shamir.

[44] **TIMESETS—A New Method for Temporal Reasoning About Programs**. *Logics of Programs*, Dexter Kozen editor, Springer-Verlag Lecture Notes in Computer Science Volume 131 (1982), 177–196.

Pnueli's introduction of temporal logic in 1977 led to an explosion of attempts to find new logics for specifying and reasoning about concurrent systems. Everyone was looking for the silver-bullet logic that would solve the field's problems. This paper is proof that I was not

immune to this fever. For reasons explained in the discussion of [50], it is best forgotten. Some people may find this of historical interest because it is an early example of an interval logic.

[45] **Byzantine Generals and Transaction Commit Protocols** (with Michael Fischer). Unpublished (April 1982).

I visited Michael Fischer at Yale in the spring of 1982. It was known that solutions to the Byzantine generals problem that can handle n Byzantine failures require $n+1$ rounds of communication. While I was at Yale, Fischer and I proved that this number of rounds were needed even to handle more benign failures.

On the trip back home to California, I got on an airplane at Laguardia Airport in the morning with a snowstorm coming in. I got off the airplane about eight hours later, still at Laguardia Airport, having written this paper. I then sent it to Fischer for his comments. I waited about a year and a half. By the time he finally decided that he wasn't going to do any more work on the paper, subsequent work by others had been published that superseded it.

[46] **The Byzantine Generals Problem** (with Marshall Pease and Robert Shostak). *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 382–401.

I have long felt that, because it was posed as a cute problem about philosophers seated around a table, Dijkstra's dining philosopher's problem received much more attention than it deserves. (For example, it has probably received more attention in the theory community than the readers/writers problem, which illustrates the same principles and has much more practical importance.) I believed that the problem introduced in [41] was very important and deserved the attention of computer scientists. The popularity of the dining philosophers problem taught me that the best way to attract attention to a problem is to present it in terms of a story.

There is a problem in distributed computing that is sometimes called the Chinese Generals Problem, in which two generals have to come to a common agreement on whether to attack or retreat, but can communicate only by sending messengers who might never arrive. I stole the idea of the generals and posed the problem in terms of a group of generals, some of whom may be traitors, who have to reach a common decision. I wanted to assign the generals a nationality that would not offend any readers. At the time, Albania was a completely

closed society, and I felt it unlikely that there would be any Albanians around to object, so the original title of this paper was *The Albanian Generals Problem*. Jack Goldberg was smart enough to realize that there were Albanians in the world outside Albania, and Albania might not always be a black hole, so he suggested that I find another name. The obviously more appropriate Byzantine generals then occurred to me.

The main reason for writing this paper was to assign the new name to the problem. But a new paper needed new results as well. I came up with a simpler way to describe the general $3n+1$-processor algorithm. (Shostak's 4-processor algorithm was subtle but easy to understand; Pease's generalization was a remarkable tour de force.) We also added a generalization to networks that were not completely connected. (I don't remember whose work that was.) I also added some discussion of practical implementation details.

[47] **Proving Liveness Properties of Concurrent Programs** (with Susan Owicki). *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 455–495.

During the late 70s and early 80s, Susan Owicki and I worked together quite a bit. We were even planning to write a book on concurrent program verification. But this paper is the only thing we ever wrote together.

In [23], I had introduced a method of proving eventuality properties of concurrent programs by drawing a lattice of predicates, where arrows from a predicate P to predicates Q_1, ..., Q_n mean that, if the program reaches a state satisfying P, it must thereafter reach a state satisfying one of the Q_i. That method never seemed practical; formalizing an informal proof was too much work.

Pnueli's introduction of temporal logic allowed the predicates in the lattice to be replaced by arbitrary temporal formulas. This turned lattice proofs into a useful way of proving liveness properties. It permitted a straightforward formalization of a particularly style of writing the proofs. I still use this proof style to prove leads-to properties, though the proofs are formalized with TLA (see [102]). However, I no longer bother drawing pictures of the lattices. This paper also introduced *at*, *in*, and *after* predicates for describing program control.

It's customary to list authors alphabetically, unless one contributed significantly more than the other, but at the time, I was unaware of this custom. Here is Owicki's account of how the ordering of the

authors was determined.

> As I recall it, you raised the question of order, and I proposed alphabetical order. You declined—I think you expected the paper to be important and didn't think it would be fair to get first authorship on the basis of a static property of our names. On the night we finished the paper, we went out to dinner to celebrate, and you proposed that if the last digit of the bill was even (or maybe odd), my name would be first. And, indeed, that's the way it came out.

[48] **An Assertional Correctness Proof of a Distributed Program**. *Science of Computer Programming 2*, 3 (December 1982), 175–206.

I showed in [27] that there is no invariant way of defining the global state of a distributed system. Assertional methods, such as [23], reason about the global state. So, I concluded that these methods were not appropriate for reasoning about distributed systems. When I wrote this paper, I was at SRI and partly funded by a government contract for which we had promised to write a correctness proof of a distributed algorithm. I tried to figure out how to write a formal proof without reasoning about the global state, but I couldn't. The final report was due, so I decided that there was no alternative to writing an assertional proof. I knew there would be no problem writing such a proof, but I expected that, with its reliance on an arbitrary global state, the proof would be ugly. To my surprise, I discovered that the proof was quite elegant. Philosophical considerations told me that I shouldn't reason about global states, but this experience indicated that such reasoning worked fine. I have always placed more reliance on experience than philosophy, so I have written assertional proofs of distributed systems ever since. (Others, more inclined to philosophy, have spent decades looking for special ways to reason about distributed systems.)

[49] **Reasoning About Nonatomic Operations**. *Proceedings of the Tenth ACM Symposium on Principles of Programming Languages*, ACM SIGACT-SIGPLAN (January 1983), 28–37.

From the time I discovered the bakery algorithm (see [12]), I was fascinated by the problem of reasoning about a concurrent program without having to break it into indivisible atomic actions. In [33], I described how to do this for behavioral reasoning. But I realized that assertional reasoning, as described in [23], was the only proof

method that could scale to more complex problems. This paper was my first attempt at assertional reasoning about nonatomic operations. It introduces the *win* (weakest invariant) operator that later appeared in [86], but using the notation of Pratt's dynamic logic rather than Dijkstra's predicate transformers.

I have in my files a letter from David Harel, who was then an editor of *Information and Control*, telling me that the paper was accepted by the journal, after revision to satisfy some concerns of the referees. I don't remember why I didn't submit a revised version. I don't think I found the referees' requests unreasonable. It's unlikely that I abandoned the paper because I had already developed the method in [86], since that didn't appear as a SRC research report until four years later. Perhaps I was just too busy.

[50] **Specifying Concurrent Program Modules**. *ACM Transactions on Programming Languages and Systems 5*, 2 (April 1983), 190–222.

The early methods for reasoning about concurrent programs dealt with proving that a program satisfied certain properties—usually invariance properties. But, proving particular properties showed only that the program satisfied those properties. There remained the possibility that the program was incorrect because it failed to satisfy some other properties. Around the early 80s, people working on assertional verification began looking for ways to write a complete specification of a system. A specification should say precisely what it means for the system to be correct, so that if we prove that the system meets its specification, then we can say that the system really is correct. (Process algebraists had already been working on that problem since the mid-70s, but there was—and I think still is—little communication between them and the assertional verification community.)

At SRI, we were working on writing temporal logic specifications. One could describe properties using temporal logic, so it seemed very natural to specify a system by simply listing all the properties it must satisfy. Richard Schwartz, Michael Melliar-Smith, and I collaborated on a paper titled *Temporal Logic Specification of Distributed Systems*, which was published in the Proceedings of the 2nd International Conference on Distributed Computing Systems, held in Paris in 1981. However, I became disillusioned with temporal logic when I saw how Schwartz, Melliar-Smith, and Fritz Vogt were spending days trying to specify a simple FIFO queue—arguing over whether the properties they listed were sufficient. I realized that, despite its aesthetic appeal,

writing a specification as a conjunction of temporal properties just didn't work in practice. So, I had my name removed from the paper before it was published, and I set about figuring out a practical way to write specifications. I came up with the approach described in this paper, which I later called the *transition axiom* method. Schwartz stopped working on specification and verification in the mid-80s. He wrote recently (in June 2000):

> [T]he same frustration with the use of temporal logic led Michael, Fritz Vogt and me to come up with Interval Logic as a higher level model in which to express time-ordered properties of events. [See [44].] As you recall, interval logic expressed constraints forward and backward around significant events in order to more closely capture the way that people describe event-driven behavior. Ultimately, I remain unsatisfied with any of our attempts, from the standpoint of reaching practical levels.

This paper is the first place I used the idea of describing a state transition as a boolean-valued function of primed and unprimed variables. However, by the early 80s, the idea must have been sufficiently obvious that I didn't claim any novelty for it, and I forgot that I had even used it in this paper until years later (see the discussion of [102]).

[51] **Specification and Proof of a Fault-Tolerant Real-Time Algorithm**. In *Highly Dependable Distributed Systems*, final report for SRI Project 4180 (Contract Number DAEA18-81-G-0062) (June 1983).

In the spring of 1983, I was called upon to contribute a chapter for the final report on a project at SRI. I chose to write a specification and correctness proof of a Byzantine general's algorithm—a distributed, real-time algorithm. (Nonfaulty components must satisfy real-time constraints, and the correctness of the algorithm depends on these constraints.) I began the exercise on a Wednesday morning. By noon that Friday, I had the final typeset output. I presume there are lots of errors; after finishing it, I never reread it carefully and I have no indication that anyone else did either. But, I have no reason to doubt the basic correctness of the proof. I never published this paper because it didn't seem worth publishing. The only thing I find remarkable about it is that so many computer scientists are unaware that, even in 1983, writing a formal correctness proof of a distributed real-time algorithm was an unremarkable feat.

[52] **The Weak Byzantine Generals Problem**. *Journal of the Association for Computing Machinery 30*, 3 (July 1983), 668–676.

This paper introduces a weaker version of the Byzantine generals problem described in [41]. The problem is "easier" because there exist approximate solutions with fewer than $3n$ processes that can tolerate n faults, something shown in [41] to be impossible for the original Byzantine generals problem. I don't remember how I came to consider this problem.

[53] **PHIL: A Semantic Structural Graphical Editor** (with Joseph Goguen). SRI International Technical Report (August 1983).

SRI had a contract with Philips to design a graphical editor for structured documents (such as programs). Goguen and I were the prime instigators and principal investigators of the project. This is the project's final report. Rindert Schutten of Philips visited SRI and implemented a very preliminary version. I felt that our design was neither novel enough to constitute a major contribution nor modest enough to be the basis for a practical system at that time, and I thought the project had been dropped. However, Goguen informed me much later that some version of the system was still being used in the early 90s, and that it had evolved into a tool for VLSI layout, apparently called MetaView.

[54] **What Good Is Temporal Logic?**. *Information Processing 83,* R. E. A. Mason, ed., Elsevier Publishers (1983), 657–668.

This was an invited paper. It describes the state of my views on specification and verification at the time. It is notable for introducing the idea of invariance under stuttering and explaining why it's a vital attribute of a specification logic. It is also one of my better-written papers.

[55] **Using Time Instead of Timeout for Fault-Tolerant Distributed Systems**. *ACM Transactions on Programming Languages and Systems 6*, 2 (April 1984), 254–280.

The genesis of this paper was my realization that, in a multiprocess system with synchronized clocks, the absence of a message can carry information. I was fascinated by the idea that a process could communicating zillions of bits of information by *not* sending messages. The practical implementation of Byzantine generals algorithms described in [46] could be viewed as an application of this idea. I used the idea as

something of a gimmick to justify the paper. The basic message of this paper should have been pretty obvious: the state machine approach, introduced in [27], allows us to turn any consensus algorithm into a general method for implementing distributed systems; the Byzantine generals algorithms of [46] were fault-tolerant consensus algorithms; hence, we had fault-tolerant implementations of arbitrary distributed systems. I published the paper because I had found few computer scientists who understood this.

[56] **The Hoare Logic Of CSP, and All That** (with Fred Schneider). *ACM Transactions on Programming Languages and Systems 6*, 2 (April 1984), 281–296.

I felt that in [40], I had presented the right way to do assertional (also known as Owicki-Gries style) reasoning about concurrent programs. However, many people were (and perhaps still are) hung up on the individual details of different programming languages and are unable to understand that the same general principles apply to all of them. In particular, people felt that "distributed" languages based on rendezvous or message passing were fundamentally different from the shared-variable language that was considered in [40]. For example, some people made the silly claim that the absence of shared variables made it easier to write concurrent programs in CSP than in more conventional languages. (My response is the equally silly assertion that it's harder to write concurrent programs in CSP because the control state is shared between processors.)

Schneider agreed with me that invariance was the central concept in reasoning about concurrent programs. He was also an expert on all the different flavors of message passing that had been proposed. We demonstrated in this paper that the basic approach of [40] worked just was well with CSP; and we claimed (without proof) that it also worked in other "distributed" languages. I found it particularly funny that we should be the ones to give a Hoare logic to CSP, while Hoare was using essentially behavioral methods to reason about CSP programs. I'm still waiting for the laughter.

[57] **Byzantine Clock Synchronization** (with Michael Melliar-Smith). *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing* (August, 1984), 68–74.

This is the preliminary conference version of [62].

[58] **Solved Problems, Unsolved Problems and NonProblems in Concurrency**. *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing* (August, 1984) 1–11.

This is the invited address I gave at the 1983 PODC conference, which I transcribed from a tape recording of my presentation. The first few minutes of the talk were not taped, so I had to reinvent the beginning. This talk is notable because it marked the rediscovery by the computer science community of Dijkstra's 1974 *CACM* paper that introduced the concept of self-stabilization. A self-stabilizing system is one that, when started in any state, eventually "rights itself" and operates correctly. The importance of self-stabilization to fault tolerance was obvious to me and a handful of people, but went completely over the head of most readers. Dijkstra's paper gave little indication of the practical significance of the problem, and few people understood its importance. So, this gem of a paper had disappeared without a trace by 1983. My talk brought Dijkstra's paper to the attention of the PODC community, and now self-stabilization is a regular subfield of distributed computing. I regard the resurrection of Dijkstra's brilliant work on self-stabilization to be one of my greatest contributions to computer science.

The paper contains one figure—copied directly from a transparency—with an obviously bogus algorithm. I tried to recreate an algorithm from memory and wrote complete nonsense. It's easy to make such a mistake when drawing a transparency, and I probably didn't bother to look at it when I prepared the paper. To my knowledge, it is the only incorrect algorithm I have published.

[59] **On a "Theorem" of Peterson**. Unpublished (October, 1984).

This three-page note gives an example that appears to contradict a theorem in a *TOPLAS* article by Gary Peterson. Whether or not it does depends on the interpretation of the statement of the theorem, which is given only informally in English. I draw the moral that greater rigor is needed. When I sent this paper to Peterson, he strongly objected to it. I no longer have his message and don't remember exactly what he wrote, but I think he said that he knew what the correct interpretation was and that I was unfairly suggesting that his theorem might be incorrect. So, I never published this note.

[60] **Buridan's Principle**. Unpublished (December 1984).

I have observed that the arbiter problem, discussed in [22], occurs in

daily life. Perhaps the most common example is when I find myself unable to decide for a fraction of a second whether to stop for a traffic light that just turned yellow or to go through. I suspect that it is actually a cause of serious accidents, and that people do drive into telephone poles because they can't decide in time whether to go to the left or the right.

A little research revealed that psychologists are totally unaware of the phenomenon. I found one paper in the psychology literature on the time taken by subjects to choose between two alternatives based on how nearly equal they were. The author's theoretical calculation yielded a formula with a singularity at zero, as there should be. He compared the experimental data with this theoretical curve, and the fit was perfect. He then drew, as the curve fitting the data, a bounded continuous graph. The singularity at zero was never mentioned in the paper.

I feel that the arbiter problem is important and should be made known to scientists outside the field of computing. So I wrote this paper, which describes the problem in its classical formulation as the problem of Buridan's ass—an ass that starves to death because it is placed equidistant between two bales of hay and has no reason to prefer one to the other. Philosophers have discussed Buridan's ass for centuries, but it apparently never occurred to any of them that the planet is not littered with dead asses only because the probability of the ass being in just the right spot is infinitesimal.

So, I wrote this paper for the general scientific community. I probably could have published it in some computer journal, but that wasn't the point. I submitted it first to *Science*. The four reviews ranged from "This well-written paper is of major philosophical importance" to "This may be an elaborate joke." One of the other reviews was more mildly positive, and the fourth said simply "My feeling is that it is rather superficial." The paper was rejected.

Some time later, I submitted the paper to *Nature*. I don't like the idea of sending the same paper to different journals hoping that someone will publish it, and I rarely resubmit a rejected paper elsewhere. So, I said in my submission letter that it had been rejected by *Science*. The editor read the paper and sent me some objections. I answered his objections, which were based on reasonable misunderstandings of the paper. In fact, they made me realize that I should explain things differently for a more general audience. He then replied with further objections of a similar nature. Throughout this exchange, I wasn't

sure if he was taking the matter seriously or if he thought I was some sort of crank. So, after answering his next round of objections, I wrote that I would be happy to revise the paper in light of this discussion if he would then send it out for review, but that I didn't want to continue this private correspondence. The next letter I received was from another *Nature* editor saying that the first editor had been reassigned and that he was taking over my paper. He then raised some objections to the paper that were essentially the same as the ones raised initially by the first editor. At that point, I gave up in disgust.

I still think that this paper is worth publishing for a general scientific audience. Among other things, it has a nice analysis of a quantum-mechanical arbiter. However, I have no idea where to publish it.

My problems in trying to publish this paper are part of a long tradition. According to one story I've heard (but haven't verified), someone at G. E. discovered the phenomenon in computer circuits in the early 60s, but was unable to convince his managers that there was a problem. He published a short note about it, for which he was fired. Charles Molnar, one of the pioneers in the study of the problem, reported the following in a lecture given on February 11, 1992, at HP Corporate Engineering in Palo Alto, California:

> One reviewer made a marvelous comment in rejecting one of the early papers, saying that if this problem *really* existed it would be so important that everybody knowledgeable in the field would have to know about it, and "I'm an expert and I don't know about it, so therefore it must not exist."

[61] **The Mutual Exclusion Problem—Part I: A Theory of Interprocess Communication, Part II: Statement and Solutions**. *Journal of the Association for Computing Machinery 33*, 2 (January 1985) 313–348.

For some time I had been looking for a mutual exclusion algorithm that satisfied my complete list of desirable properties. I finally found one—the $N!$-bit algorithm described in this paper. The algorithm is wildly impractical, requiring $N!$ bits of storage for N processors, but practicality was not one of my requirements. So, I decided to publish a compendium of everything I knew about the theory of mutual exclusion.

The 3-bit algorithm described in this paper came about because of a visit by Michael Rabin. He is an advocate of probabilistic algorithms,

and he claimed that a probabilistic solution to the mutual exclusion problem would be better than a deterministic one. I believe that it was during his brief visit that we came up with a probabilistic algorithm requiring just three bits of storage per processor. Probabilistic algorithms don't appeal to me. (This is a question of aesthetics, not practicality.) So later, I figured out how to remove the probability and turn it into a deterministic algorithm.

The first part of the paper covers the formalism for describing nonatomic operations that I had been developing since the 70s, and that is needed for a rigorous exposition of mutual exclusion. (See the discussion of [70].)

[62] **Synchronizing Clocks in the Presence of Faults** (with Michael Melliar-Smith). *Journal of the Association for Computing Machinery 32*, 1 (January 1985), 52–78.

Practical implementation of Byzantine agreement requires synchronized clocks. For an implementation to tolerate Byzantine faults, it needs a clock synchronization algorithm that can tolerate those faults. When I arrived at SRI, there was a general feeling that we could synchronize clocks by just having each process use a Byzantine agreement protocol to broadcast its clock value. I was never convinced by that hand waving. So, at some point I tried to write down precise clock-synchronization algorithms and prove their correctness. The two basic Byzantine agreement algorithms from [46] did generalize to clock-synchronization algorithms. In addition, Melliar-Smith had devised the interactive convergence algorithm, which is also included in the paper. (As I recall, that algorithm was his major contribution to the paper, and I wrote all the proofs.)

Writing the proofs turned out to be much more difficult than I had expected (see [27]). I worked very hard to make them as short and easy to understand as I could. So, I was rather annoyed when a referee said that the proofs seemed to have been written quickly and could be simplified with a little effort. In my replies to the reviews, I referred to that referee as a "supercilious bastard". Some time later, Nancy Lynch confessed to being that referee. She had by then written her own proofs of clock synchronization and realized how hard they were.

Years later, John Rushby and his colleagues at SRI wrote mechanically verified versions of my proofs. They found only a couple of minor errors. I'm rather proud that, even before I knew how to write reliable, structured proofs (see [101]), I was sufficiently careful and disciplined

to have gotten those proofs essentially correct.

[63] **What It Means for a Concurrent Program to Satisfy a Specification: Why No One Has Specified Priority**. *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, ACM SIGACT-SIGPLAN (January 1985), 78–83.

I must have spent a lot of time at SRI arguing with Schwartz and Melliar-Smith about the relative merits of temporal logic and transition axioms. (See the discussion of [50].) I don't remember exactly what happened, but this paper's acknowledgment section says that "they kept poking holes in my attempts to specify FCFS [first-come, first-served] until we all finally recognized the fundamental problem [that it can't be done]."

[64] **Constraints: A Uniform Approach to Aliasing and Typing** (with Fred Schneider). *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, ACM SIGACT-SIGPLAN (January 1985), 205–216.

My generalized Hoare logic requires reasoning about control predicates, using the *at*, *in*, and *after* predicates introduced in [47]. These are not independent predicates—for example, being after one statement is synonymous with being at the following statement. At some point, Schneider and I realized that the relations between control predicates could be viewed as a generalized form of aliasing. Our method of dealing with control predicates led to a general approach for handling aliasing in ordinary Hoare logic, which is described in this paper. In addition to handling the usual aliasing in programs, our method allowed one to declare that the variables r and θ were aliases of the variables x and y according to the relations $x = r\cos\theta$ and $y = r\sin\theta$.

We generalized this work to handle arrays and pointers, and even cited a later paper about this generalization. But, as has happened so often when I write a paper that mentions a future one, the future paper was never written.

[65] **Recursive Compiling and Programming Environments (Summary)**. Rejected from the 1985 POPL Conference..

This is an extended abstract I submitted to the 1995 POPL conference. (I never wrote the complete version.) It proposes the idea of recursive compiling, in which a program constructs a text string and calls the compiler to compile it in the context of the current program environ-

ment. Thus, a variable *foo* in the string is interpreted by the compiler to mean whatever *foo* means at the current point in the calling program. No one found the idea very compelling. When I discussed it with Eric Roberts, he argued that run-time linking would be a simpler way to provide the same functionality. I don't know if Java's reflection mechanism constitutes recursive compiling or just run-time linking.

[66] **Distributed Snapshots: Determining Global States of a Distributed System** (with Mani Chandy). *ACM Transactions on Computer Systems 3*, 1 (February, 1985), 63–75.

The distributed snapshot algorithm described here came about when I visited Chandy, who was then at the University of Texas in Austin. He posed the problem to me over dinner, but we had both had too much wine to think about it right then. The next morning, in the shower, I came up with the solution. When I arrived at Chandy's office, he was waiting for me with the same solution. I consider the algorithm to be a straightforward application of the basic ideas from [27].

[67] **Formal Foundation for Specification and Verification** (with Fred Schneider). Chapter 5 in *Distributed Systems: Methods and Tools for Specification*, Alford et al. Lecture Notes in Computer Science, Number 190. Springer-Verlag, Berlin (1985).

This volume contains the notes for a two-week course given in Munich in April of 1984 and again in April of 1985. Fred Schneider and I lectured on the contents of [56] and [47]. This chapter is of historical interest because it's the first place where I published the precise definition of a safety property. (The concepts of safety and liveness were introduced informally in [23].) This inspired Schneider to think about what the precise definition of liveness might be. Shortly thereafter, he and Bowen Alpern came up with the formal definition.

[68] **An Axiomatic Semantics of Concurrent Programming Languages**. In *Logics and Models of Concurrent Systems*, Krzysztof Apt, editor. Springer-Verlag, Berlin (1985), 77–122.

This paper appeared in a workshop held in Colle-sur-Loup, in the south of France, in October, 1984. There is a long history of work on the semantics of programming languages. When people began studying concurrency in the 70s, they naturally wrote about the semantics of concurrent languages. It always seemed to me that defining the semantics of a concurrent language shouldn't be very hard. Once you

know how to specify a concurrent system, it's a straightforward task to give a semantics to a concurrent programming language by specifying the programs written in it. Writing this paper allowed me to demonstrate that writing a semantics is as easy as I thought it was. What I did discover from writing the paper is that the semantics of programming languages is a very boring subject. I found this paper boring to write; I find it boring to read. I have never worked on the semantics of programming languages again.

[69] **LaTeX: A Document Preparation System**. *Addison-Wesley, Reading, Mass. (1986).*

In the early 80s, I was planning to write the Great American Concurrency Book. I was a TeX user, so I would need a set of macros. I thought that, with a little extra effort, I could make my macros usable by others. Don Knuth had begun issuing early releases of the current version of TeX, and I figured I could write what would become its standard macro package. That was the beginning of LaTeX. I was planning to write a user manual, but it never occurred to me that anyone would actually pay money for it. In 1983, Peter Gordon, an Addison-Wesley editor, and his colleagues visited me at SRI. Here is his account of what happened.

> Our primary mission was to gather information for Addison-Wesley "to publish a computer-based document processing system specifically designed for scientists and engineers, in both academic and professional environments." This system was to be part of a series of related products (software, manuals, books) and services (database, production). (La)TeX was a candidate to be at the core of that system. (I am quoting from the original business plan.) Fortunately, I did not listen to your doubt that anyone would buy the LaTeX manual, because more than a few hundred thousand people actually did. The exact number, of course, cannot accurately be determined, inasmuch as many people (not all friends and relatives) bought the book more than once, so heavily was it used.

Meanwhile, I still haven't written the Great American Concurrency Book.

[70] **On Interprocess Communication—Part I: Basic Formalism,**

Part II: Algorithms. *Distributed Computing 1*, 2 (1986), 77–101. Also appeared as SRC Research Report 8.

Most computer scientists regard synchronization problems, such as the mutual exclusion problem, to be problems of mathematics. How can you use one class of mathematical objects, like atomic reads and writes, to implement some other mathematical object, like a mutual exclusion algorithm? I have always regarded synchronization problems to be problems of physics. How do you use physical objects, like registers, to achieve physical goals, like not having two processes active at the same time?

With the discovery of the bakery algorithm (see [12]), I began considering the question of how two processes communicate. I came to the conclusion that asynchronous communication requires some object whose state can be changed by one process and observed by the other. We call such an object a register. This paper introduced three classes of registers. The weakest class with which arbitrary synchronization is possible is called *safe*. The next strongest is called *regular* and the strongest, generally assumed by algorithm writers, is called *atomic*.

I had obtained all the results presented here in the late 70s and had described them to a number of computer scientists. Nobody found them interesting, so I never wrote them up. Around 1984, I saw a paper by Jay Misra, motivated by VLSI, that was heading in the general direction of my results. It made me realize that, because VLSI had started people thinking about synchronization from a more physical perspective, they might now be interested in my results about registers. So, I wrote this paper. As with [61], the first part describes my formalism for describing systems with nonatomic operations. This time, people were interested—perhaps because it raised the enticing unsolved problem of implementing multi-reader and multi-writer atomic registers. It led to a brief flurry of atomic register papers.

Fred Schneider was the editor who processed this paper. He kept having trouble understanding the proof of my atomic register construction. After a couple of rounds of filling in the details of the steps that Schneider couldn't follow, I discovered that the algorithm was incorrect. Fortunately, I was able to fix the algorithm and write a proof that he, I, and, as far as I know, all subsequent readers did believe.

Some fifteen years later, Jerry James, a member of the EECS department at the University of Kansas, discovered a small error in Proposition 1 when formalizing the proofs with the PVS mechanical

verification system. He proved a corrected version of the proposition and showed how that version could be used in place of the original one.

[71] **The Byzantine Generals** (with Danny Dolev, Marshall Pease, and Robert Shostak). In *Concurrency Control and Reliability in Distributed Systems*, Bharat K. Bhargava, editor, Van Nostrand Reinhold (1987) 348–369.

I have only a vague memory of this paper. I believe Bhargava asked me to write a chapter about the results in [41] and [46]. I was probably too lazy and asked Dolev to write a chapter that combined his more recent results on connectivity requirements with our original results. I would guess that he did all the work, though I must have at least read and approved of what he wrote.

[72] **A Formal Basis for the Specification of Concurrent Systems**. In *Distributed Operating Systems: Theory and Practice*, Paker, Banatre and Bozyiğit, editors, Springer-Verlag (1987), 1–46.

This paper describes the transition axiom method I introduced in [50]. It was written for a NATO Advanced Study Institute that took place in Turkey in August, 1986, and contains little that was new.

[73] **A Fast Mutual Exclusion Algorithm**. *ACM Transactions on Computer Systems 5*, 1 (February 1987), 1–11. Also appeared as SRC Research Report 7.

Soon after I arrived at SRC, I was approached by some people at WRL (Digital's Western Research Laboratory) who were building a multiprocessor computer. They wanted to avoid having to add synchronization instructions, so they wanted to know how efficiently mutual exclusion could be implemented with just read and write instructions. They figured that, with properly designed programs, contention for a critical section should be rare, so they were interested in efficiency in the absence of contention. I deduced the lower bound on the number of operations required and the optimal algorithm described in this paper. They decided that it was too slow, so they implemented a test-and-set instruction.

I find it remarkable that, 20 years after Dijkstra first posed the mutual exclusion problem, no one had thought of trying to find solutions that were fast in the absence of contention. This illustrates why I like working in industry: the most interesting theoretical problems

come from implementing real systems.

[74] **Derivation of a Simple Synchronization Algorithm**. Rejected by *Information Processing Letters* (February 1987).

Chuck Thacker posed a little synchronization problem to me, which I solved with Jim Saxe's help. At that time, deriving concurrent algorithms was the fashion—the idea that you discover the algorithm by some form of black magic and then verify it was considered passé. So, I decided to see if I could have derived the algorithm from approved general principles. I discovered that I could—at least, informally—and that this informal derivation seemed to capture the thought process that led me to the solution in the first place.

[75] **Distribution**. Email message sent to a DEC SRC bulletin board at 12:23:29 PDT on 28 May 87.

This message is the source of the following observation, which has been quoted (and misquoted) rather widely:

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

[76] **Document Production: Visual or Logical?**. *Notices of the American Mathematical Society* (June 1987), 621–624.

Richard Palais ran a column on mathematical typesetting in the *AMS Notices*, and he invited me to be guest columnist. This is what I wrote—a short exposition of my ideas about producing mathematical documents.

[77] **Synchronizing Time Servers**. SRC Research Report 18 (June 1987).

When I joined DEC in 1985, they were the world leader in networking. Using their VMS operating system, I could type a simple *copy* command to a computer in California, specifying a file and machine name, to copy a file from a computer in Massachusetts. Even today, I can't copy a file from Massachusetts to California nearly as easily with Unix or Windows.

The people responsible for DEC's network systems were the Network and Communications group (NAC). Around 1987, NAC asked for my help in designing a network time service. I decided that there

were two somewhat conflicting requirements for a time service: delivering the correct time, and keeping the clocks on different computers closely synchronized. This paper describes the algorithms I devised for doing both.

I withdrew the paper because Tim Mann observed that the properties I proved about the algorithms were weaker than the ones needed to make them interesting. The major problem is that the algorithms were designed to guarantee both a bound ϵ on the synchronization of each clock with a source of correct time and an independent bound δ on the synchronization between any two clocks that could be made much smaller than ϵ. Mann observed that the bound I proved on δ was not the strong one independent of ϵ that I had intended to prove. We believe that the algorithms do satisfy the necessary stronger properties, and Mann and I began rewriting the paper with the stronger results. But that paper is still only partly written and is unlikely ever to see the light of day.

[78] **Control Predicates Are Better than Dummy Variables for Representing Program Control**. *ACM Transactions on Programming Languages and Systems 10*, 2 (April 1988), 267–281. Also appeared as SRC Research Report 11.

This paper describes an example I came across in which the explicit control predicates introduced in [47] lead to a simpler proof than do dummy variables. This example served as an excuse. The real reason for publishing it was to lobby for the use of control predicates. There used to be an incredible reluctance by theoretical computer scientists to mention the control state of a program. When I first described the work in [40] to Albert Meyer, he immediately got hung up on the control predicates. We spent an hour arguing about them—I saying that they were necessary (as was first proved by Susan in her thesis), and he saying that I must be doing something wrong. I had the feeling that I was arguing logical necessity against religious belief, and there's no way logic can overcome religion.

[79] **"EWD 1013"**. Unpublished. Probably written around April, 1988.

Dijkstra's EWD 1013, *Position Paper on "Fairness"*, argues that fairness is a meaningless requirement because it can't be verified by observing a system for a finite length of time. The weakness in this argument is revealed by observing that it applies just as well to termination. To make the point, I wrote this note, which claims to be an

early draft of EWD 1013 titled *Position Paper on "Termination"*. It is, of course, essentially the same as EWD 1013 with *fairness* replaced by *termination*. Because of other things that happened at that time, I was afraid that Dijkstra might not take it in the spirit of good fun in which it was intended, and that he might find it offensive. So, I never showed it to anyone but a couple of friends. I think the passage of time has had enough of a mellowing effect that no one will be offended any more by it. It is now of more interest for the form than for the content.

[80] **Another Position Paper on Fairness** (with Fred Schneider). *Software Engineering Notes 13*, 3 (July, 1988) 1–2.

This is a more traditional response to Dijkstra's EWD 1013 (see [79]). We point out that Dijkstra's same argument can be applied to show that termination is a meaningless requirement because it can't be refuted by looking at a program for a finite length of time. The real argument in favor of fairness, which we didn't mention, is that it is a useful concept when reasoning about concurrent systems.

[81] **A Lattice-Structured Proof of a Minimum Spanning Tree Algorithm** (with Jennifer Welch and Nancy Lynch). *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing* (August, 1988).

In 1983, Gallager, Humblet, and Spira published a distributed algorithm for computing a minimum spanning tree. For several years, I regarded it as a benchmark problem for verifying concurrent algorithms. A couple of times, I attempted to write an invariance proof, but the invariant became so complicated that I gave up. On a visit to M.I.T., I described the problem to Nancy Lynch, and she became interested in it too. I don't remember exactly how it happened, but we came up with the idea of decomposing the proof not as a simple hierarchy of refinements, but as a lattice of refinements. Being an efficient academic, Lynch got Jennifer Welch to do the work of actually writing the proof as part of her Ph. D. thesis. This paper is the conference version, written mostly by her.

There were three proofs of the minimum spanning-tree algorithm presented at PODC that year: ours, one by Willem-Paul de Roever and his student Frank Stomp, and the third by Eli Gafni and his student Ching-Tsun Chou. Each paper used a different proof method. I thought that the best of the three was the one by Gafni and Chou—not

because their proof method was better, but because they understood the algorithm better and used their understanding to simplify the proof. If they had tried to formalize their proof, it would have turned into a standard invariance proof. Indeed, Chou eventually wrote such a formal invariance proof in his doctoral thesis.

The Gallager, Humblet, and Spira algorithm is complicated and its correctness is quite subtle. (Lynch tells me that, when she lectured on its proof, Gallager had to ask her why it works in a certain case.) There doesn't seem to be any substitute for a standard invariance proof for this kind of algorithm. Decomposing the proof the way we did seemed like a good idea at the time, but in fact, it just added extra work. (See [124] for a further discussion of this.)

[82] **A Simple Approach to Specifying Concurrent Systems**. *Communications of the ACM 32*, 1 (January 1989), 32–45. Also appeared as SRC Research Report 15.

This is a "popular" account of the transition-axiom method that I introduced in [50]. To make the ideas more accessible, I wrote it in a question-answer style that I copied from the dialogues of Galileo. The writing in this paper may be the best I've ever done.

[83] **Pretending Atomicity** (with Fred Schneider). SRC Research Report 44 (May 1989).

Reasoning about concurrent systems is simpler if they have fewer separate atomic actions. To simplify reasoning about systems, we'd like to be able to combine multiple small atomic actions into a single large one. This process is called *reduction*. This paper contains a reduction theorem for multiprocess programs. It was accepted for publication, subject to minor revisions, in *ACM Transactions on Programming Languages and Systems*. However, after writing it, I invented TLA, which enabled me to devise a stronger and more elegant reduction theorem. Schneider and I began to revise the paper in terms of TLA. We were planning to present a weaker, simpler version of the TLA reduction theorem that essentially covered the situations considered in this report. However, we never finished that paper. A more general TLA reduction theorem was finally published in [123].

[84] **Realizable and Unrealizable Specifications of Reactive Systems** (with Martín Abadi and Pierre Wolper). *Automata, Languages*

and Programming, Springer-Verlag (July 1989) 1–17.

Abadi and I came upon the concept of realizability in [97]. Several other people independently came up with the idea at around the same time, including Wolper. Abadi and Wolper worked together to combine our results and his into a single paper. Abadi recollects that the section of the paper dealing with the general case was mostly ours, and Wolper mostly developed the finite case, including the algorithms. He remembers adopting the term "realizability" from realizability in intuitionistic logic, and thinking of the relation with infinite games after seeing an article about such games in descriptive set theory in the Journal of Symbolic Logic. As I recall, I wasn't very involved in the writing of this paper.

[85] **A Temporal Logic of Actions**. SRC Research Report 47 (April 1990).

This was my first attempt at TLA, and I didn't get it quite right. It is superseded by [102].

[86] ***win* and *sin*: Predicate Transformers for Concurrency**. *ACM Transactions on Programming Languages and Systems 12*, 3 (July 1990) 396–428. Also appeared as SRC Research Report 17 (May 1987).

I had long been fascinated with algorithms that, like the bakery algorithm of [12], do not assume atomicity of their individual operations. I devised the formalism first published in [33] for writing behavioral proofs of such algorithms. I had also long been a believer in invariance proofs, which required that the algorithm be represented in terms of atomic actions. An assertional proof of the bakery algorithm required modeling its nonatomic operations in terms of multiple atomic actions—as I did in [12]. However, it's easy to introduce tacit assumptions with such modeling. Indeed, sometime during the early 80s I realized that the bakery algorithm required an assumption about how a certain operation is implemented that I had never explicitly stated, and that was not apparent in any of the proofs I had written. This paper introduces a method of writing formal assertional proofs of algorithms directly in terms of their nonatomic operations. It gives a proof of the bakery algorithm that explicitly reveals the needed assumption. However, I find the method very difficult to use. With practice, perhaps one could become facile enough with it to make it practical. However, there don't seem to be enough algorithms requiring reasoning about nonatomic actions for anyone to acquire that

facility.

[87] **A Theorem on Atomicity in Distributed Algorithms**. *Distributed Computing 4*, 2 (1990), 59–68. Also appeared as SRC Research Report 28.

This paper gives a reduction theorem for distributed algorithms (see the discussion of [83]). It includes what I believe to be the first reduction result for liveness properties.

[88] **Distributed Computing: Models and Methods** (with Nancy Lynch). *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, Jan van Leeuwen, editor, Elsevier (1990), 1157–1199.

Jan van Leeuwen asked me to write a chapter on distributed systems for this handbook. I realized that I wasn't familiar enough with the literature on distributed algorithms to write it by myself, so I asked Nancy Lynch to help. I also observed that there was no chapter on assertional verification of concurrent algorithms. (That was probably due to the handbook's geographical origins, since process algebra rules in Europe.) So I included a major section on proof methods. As I recall, I wrote most of the first three sections and Lynch wrote the fourth section on algorithms pretty much by herself.

[89] **A Completeness Theorem for TLA**. Unpublished (October, 1990).

This is a note that states and proves a relative completeness result for the axioms of TLA in the absence of temporal existential quantification (variable hiding). A text version of the complete note and of all other TLA notes are available on the web at `http://research.microsoft.com/users/lamport/tla/notes.html`. There are undoubtedly errors in the proof, but I think they're minor.

[90] **The Concurrent Reading and Writing of Clocks**. *ACM Transactions on Computer Systems 8*, 4 (November 1990), 305–310. Also appeared as SRC Research Report 27.

This paper uses the results from [25] to derive a couple of algorithms for reading and writing multi-word clocks. These algorithms are in the same vein as the ones in [25], involving reading and writing multi-digit numbers in opposite directions. In fact, I think I knew the algorithms when I wrote [25]. When the problem of reading and writing a two-word clock arose in a system being built at SRC, I was surprised to

discover that the solution wasn't in [25]. I don't know why it wasn't, but I welcomed the opportunity to publish a paper reminding people of the earlier results.

[91] **The Mutual Exclusion Problem Has Been Solved**. *Communications of the ACM 34*, 1 (January 1991), 110.

In 1990, *CACM* published one of their self-assessment procedures, this time on concurrent programming. The "correct" answer to one of the questions implied that mutual exclusion can be implemented only using atomic operations that are themselves implemented with lower-level mutual exclusion. It seemed appropriate to point out that this was wrong, and that I had actually solved the mutual exclusion problem 16 years earlier in [12]—ironically, an article in *CACM*. So, I submitted this short note to that effect. The quotation from Samuel Johnson at the end is one that Bob Taylor likes very much and taught to everyone at SRC when he was lab director.

The original version, which I no longer have, quoted all the sources cited in support of the "correct" answer, showing how all those experts in the field had no idea that the mutual exclusion problem had been solved. However, the editor of *CACM* took it upon himself to handle my submission personally. He insisted that all this material be deleted, along with the accompanying sarcasm. Although I didn't like this editorial bowdlerization, I didn't feel like fighting.

I was appalled at how this note finally appeared. I have never seen a published article so effectively hidden from the reader. I defy anyone to take that issue of *CACM* and find the note without knowing the page number.

[92] **The Existence of Refinement Mappings** (with Martín Abadi). *Theoretical Computer Science 82*, 2 (May 1991), 253–284. (An abridged version appeared in *Proceedings of the Third Annual Logic In Computer Science Conference* (July 1988).) Also appeared as SRC Research Report 29.

The method of proving that one specification implements another by using a refinement mapping was well-established by the mid-80s. (It is clearly described in [54], and it also appears in [50].) It was known that constructing the refinement mapping might require adding history variables to the implementation. Indeed, Lam and Shankar essentially constructed all their refinement mappings with history variables. Jim Saxe discovered a simple example showing that history variables

weren't enough. To handle that example, he devised a more complicated refinement-mapping rule. I realized that I could eliminate that complicated rule, and use ordinary refinement, by introducing a new kind of dummy variable that I called a prophecy variable. A prophecy variable is very much like a history variable, except it predicts the future instead of remembering the past. (Nancy Lynch later rediscovered Saxe's rule and used it to "simplify" refinement proofs by eliminating prophecy variables.) I then remembered a problematic example by Herlihy and Wing in their classic paper *Linearizability: A Correctness Condition for Concurrent Objects* that could be handled with prophecy variables.

This paper was my first collaboration with Abadi. Here's my recollection of how it was written. I had a hunch that history and prophecy variables were all one needed. Abadi had recently joined SRC, and this seemed like a fine opportunity to interest him in the things I was working on. So I described my hunch to him and suggested that he look into proving it. He came back in a few weeks with the results described in the paper. My hunch was right, except that there were hypotheses needed that I hadn't suspected. Abadi, however, recalls my having had a clearer picture of the final theorem, and that we worked out some of the details together when writing the final proof.

I had just developed the structured proof style described in [101], so I insisted that we write our proofs in this style, which meant rewriting Abadi's original proofs. In the process, we discovered a number of minor errors in the proofs, but no errors in the results.

This paper won the LICS 1988 Test of Time Award (awarded in 2008).

[93] **Preserving Liveness: Comments on 'Safety and Liveness from a Methodological Point of View'** (with Martín Abadi et al.). *Information Processing Letters 40*, 3 (November 1991), 141–142.

This is a very short article—the list of authors takes up almost as much space as the text. In a note published in *IPL*, Dederichs and Weber rediscovered the concept of non-machine-closed specifications. We observed here that their reaction to those specifications was naive.

[94] **Critique of the Lake Arrowhead Three**. *Distributed Computing 6*, 1 (1992), 65–71.

For a number of years, I was a member of a committee that planned an annual workshop at Lake Arrowhead, in southern California. I

was finally pressured into organizing a workshop myself. I got Brent Hailpern to be chairman of a workshop on specification and verification of concurrent systems. A large part of the conference was devoted to a challenge problem of specifying sequential consistency. This was a problem that, at the time, I found difficult. (I later learned how to write the simple, elegant specification that appears in [126].)

Among the presentations at the workshop session on the challenge problem, there were only two serious attempts at solving the problem. (As an organizer, I felt that I shouldn't present my own solution.) After a long period of review and revision, these two and a third, subsequently-written solution, appeared in a special issue of *Distributed Computing*. This note is a critique of the three solutions that I wrote for the special issue.

[95] **The Reduction Theorem**. Unpublished (April, 1992).

This note states and proves a TLA reduction theorem. See the discussion of [123]. Text versions of this and all other TLA notes are available on the web at `http://research.microsoft.com/users/lamport/tla/notes.html`.

[96] **Mechanical Verification of Concurrent Systems with TLA** (with Urban Engberg and Peter Grønning). *Computer-Aided Verification*, G. v. Bochmann and D. K. Probst editors. (Proceedings of the Fourth International Conference, CAV'92.) Lecture Notes in Computer Science, number 663, Springer-Verlag, (June, 1992) 44–55.

When I developed TLA, I realized that, for the first time, I had a formalism that really was completely formal—so formal that mechanically checking TLA proofs should be straightforward. Working out a tiny example (the specification and trivial implementation of mutual exclusion) using the LP theorem prover, I confirmed that this was the case. I used LP mainly because we had LP experts at SRC—namely, Jim Horning and Jim Saxe.

My tiny example convinced me that we want to reason in TLA, not in LP. To do this, we need to translate a TLA proof into the language of the theorem prover. The user should write the proof in the hierarchical style of [101], and the prover should check each step. One of the advantages of this approach turned out to be that it allows separate translations for the action reasoning and temporal reasoning. This is important because about 95% of a proof consists of action reasoning, and these proofs are much simpler if done with a special

translation than in the same translation that handles temporal formulas. (In action reasoning, x and x' are two completely separate variables; x' is handled no differently than the variable y.) So, I invited Urban Engberg and Peter Grønning, who were then graduate students in Denmark, to SRC for a year to design and implement such a system. The result of that effort is described in this paper. For his doctoral research, Engberg later developed the system into one he called TLP.

Georges Gonthier demonstrated how successful this system was in his mechanical verification of the concurrent garbage collector developed by Damien Doligez and hand-proved in his thesis. Gonthier estimated that using TLP instead of working directly in LP reduced the amount of time it took him to do the proof by about a factor of five. His proof is reported in:

> Georges Gonthier, *Verifying the Safety of a Practical Concurrent Garbage Collector*, in Rajeev Alur, Thomas A. Henzinger (Ed.): *Computer Aided Verification, 8th International Conference, CAV '96*. Lecture Notes in Computer Science, Vol. 1102, Springer, 1996, 462–465.

TLP's input language was essentially a very restricted subset of TLA$^+$ (described in [127])—a language that did not exist when TLP was implemented. Extending it to handle all of TLA$^+$ would be a simple matter of programming. However, TLP is no longer maintained and probably no longer works, since it was based on an old version of the LP prover. Moreover, LP no longer seems to be the best prover to use. The next step is an industrial-strength system that accepts all of TLA$^+$ and that makes it easy to add different theorem provers as back ends, so the user can have a choice of what prover to use for each step. I am still waiting for someone to volunteer to build such a system.

[97] **Composing Specifications** (with Martín Abadi). *ACM Transactions on Programming Languages and Systems 15*, 1 (January 1993), 73–132. Also appeared as SRC Research Report 66. A preliminary version appeared in *Stepwise Refinement of Distributed Systems*, J. W. de Bakker, W.-P. de Roever, and G. Rozenberg editors, Springer-Verlag Lecture Notes in Computer Science Volume 430 (1989), 1–41..

Since the late 80s, I had vague concerns about separating the specification of a system from requirements on the environment. The ability to write specifications as mathematical formulas (first with tempo-

ral logic and then, for practical specifications, with TLA) provides an answer. The specification is simply E implies M, where E specifies what we assume of the environment and M specifies what the system guarantees. This specification allows behaviors in which the system violates its guarantee and the environment later violates its assumption—behaviors that no correct implementation could allow. So, we defined the notion of the realizable part of a specification and took as the system specification the realizable part of E implies M. We later decided that introducing an explicit *realizable-part* operator was a mistake, and that it was better to replace implication with a temporal *while* operator that made the specifications realizable. That's the approach we took in [112], which supersedes this paper.

This is the second paper in which we used structured proofs, the first being [92]. In this case, structured proofs were essential to getting the results right. We found reasoning about realizability to be quite tricky, and on several occasions we convinced ourselves of incorrect results, finding the errors only by trying to write structured proofs.

[98] **Verification of a Multiplier: 64 Bits and Beyond** (with R. P. Kurshan). *Computer-Aided Verification*, Costas Courcoubetis, editor. (Proceedings of the Fifth International Conference, CAV'93.) Lecture Notes in Computer Science, number 697, Springer-Verlag (June, 1993), 166–179.

As I observed in [124], verifying a system by first decomposing it into separate subsystems can't reduce the size of a proof and usually increases it. However, such a decomposition can reduce the amount of effort if it allows much of the resulting proof to be done automatically by a model checker. This paper shows how the decomposition theorem of [112] can be used to decompose a hardware component (in this case, a multiplier) that is too large to be verified by model checking alone. Here is Kurshan's recollection of how this paper came to be. My CAV'92 talk was mostly about [101], and the "Larch support" refers to [96]

> I cornered you after your invited address at CAV92. At CAV, you talked about indenting (and TLA, and its Larch support). I challenged you with a matter I had been thinking about since at least 1990, the year of the first CAV. In the preface to the CAV90 proceedings, I stated as a paramount challenge to the CAV community, to create a beneficial inter-

face between automated theorem proving, and model checking.

I asked you if you thought that linking TLA/Larch with S/R (which should be simple to do on account of their very close syntax and semantics for finite-state models), could be useful. I suggested the (artificial) problem of verifying a multiplier constructed from iterated stages of a very complex 8×8 multiplier. The 8×8 multiplier would be too complex to verify handily with a theorem prover. A (say) 64×64 multiplier could be built from the 8×8 one. We'd use the model checker (cospan) to verify the 8×8, and Larch/TLA to verify the induction step. You liked the idea, and we did it, you working with Urban and I working with Mark [Foissoitte].

Sadly, with the interface in place, I was unable to come up with a non-artificial feasible application. To this day, although there have been a succession of such interfaces built (I believe ours was the first), none has really demonstrated a benefit on a real application. The (revised) challenge is to find an application in which the combination finds a bug faster than either one could by itself.

[99] **Verification and Specification of Concurrent Programs**. *A Decade of Concurrency: Reflections and Perspectives*, J. W. de Bakker, W.-P. de Roever, and G. Rozenberg editors. Lecture Notes in Computer Science, number 803, Springer-Verlag, (June, 1993) 347–374.

In keeping with the theme of the workshop, this paper provides a brief, biased overview of 18 years of verifying and specifying concurrent systems, along with an introduction to TLA. Looking at it almost 10 years later, I find it a rather nice read.

[100] **Hybrid Systems in TLA$^+$**. *Hybrid Systems*, Robert L. Grossman, Anil Nerode, Hans Rischel, and Anders P. Ravn, editors. Lecture Notes in Computer Science, number 736, Springer-Verlag (1993), 77–102.

In the early 90s, hybrid systems became a fashionable topic in formal methods. Theoreticians typically respond to a new problem domain by inventing new formalisms. Physicists don't have to revise the theory of differential equations every time they study a new kind of system, and computer scientists shouldn't have to change their formalisms when

they encounter a new kind of system. Abadi and I showed in [106] that TLA can handle real-time specifications by simply introducing a variable to represent the current time. It's just as obvious that it can handle hybrid systems by introducing variables to represent other physical quantities. It is often necessary to demonstrate the obvious to people, and the opportunity to do this arose when there was a workshop in Denmark devoted to a toy problem of specifying a simple gas burner and proving the correctness of a simple implementation. I was unable to attend the workshop, but I did write this TLA$^+$ solution. (The version of TLA$^+$ used here is slightly different from the more current version that is described in [127].)

The correctness conditions given in the problem statement included an *ad hoc* set of rules about how long the gas could be on if the flame was off. The purpose of those conditions was obviously to prevent a dangerous build-up of unburned gas. To demonstrate the power of TLA$^+$, and because it made the problem more fun, I wrote a higher-level requirement stating that the concentration of gas should be less than a certain value. Assuming that the dissipation of unburned gas satisfied a simple differential equation, I proved that their conditions implied my higher-level specification—under suitable assumptions about the rate of diffusion of the gas. This required, among other things, formally specifying the Riemann integral, which took about 15 lines. I also sketched a proof of the correctness of the next implementation level. All of this was done completely in TLA$^+$. The introduction to the volume says that I "extend[ed] ... TLA$^+$ with explicit variables that denote continuous states and clocks." That, of course, is nonsense. Apparently, by their logic, you have extended C if you write a C program with variables named *time* and *flame* instead of t and f.

[101] **How to Write a Proof**. *American Mathematical Monthly* 102, 7 (August-September 1995) 600–608. Also appeared in *Global Analysis in Modern Mathematics*, Karen Uhlenbeck, editor. Publish or Perish Press, Houston. Also appeared as SRC Research Report 94.

TLA gave me, for the first time, a formalism in which it was possible to write completely formal proofs without first having to add an additional layer of formal semantics. I began writing proofs the way I and all mathematicians and computer scientists had learned to write them, using a sequence of lemmas whose proofs were a mixture of prose and formulas. I quickly discovered that this approach collapsed under the

weight of the complexity of any nontrivial proof. I became lost in a maze of details, and couldn't keep track of what had and had not been proved at any point. Programmers learned long ago that the way to handle complexity is with hierarchical structuring. So, it was quite natural to start structuring the proofs hierarchically, and I soon developed a simple hierarchical proof style. It then occurred to me that this structured proof style should be good for ordinary mathematical proofs, not just for formal verification of systems. Trying it out, I found that it was great. I now never write old-fashioned unstructured proofs for myself, and use them only in some papers for short proof sketches that are not meant to be rigorous.

I first presented these ideas in a talk at a celebration of the 60th birthday of Richard Palais, my *de jure* thesis advisor, collaborator, and friend. I was invited along with all of Palais' former doctoral students, and I was the only non-mathematician who gave a talk. (I believe all the other talks presented that day appear among the articles in the volume edited by Uhlenbeck.) Lots of people jumped on me for trying to take the fun out of mathematics. The strength of their reaction indicates that I hit a nerve. Perhaps they really do think it's fun having to recreate the proofs themselves if they want to know whether a theorem in a published paper is actually correct, and to have to struggle to figure out why a particular step in the proof is supposed to hold. I republished the paper in the *AMM Monthly* so it would reach a larger audience of mathematicians. Maybe I should republish it again for computer scientists.

[102] **The Temporal Logic of Actions**. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994), 872–923. Also appeared as SRC Research Report 79.

This paper introduces TLA, which I now believe is the best general formalism for describing and reasoning about concurrent systems. The new idea in TLA is that one can use *actions*—formulas with primed and unprimed variables—in temporal formulas. An action describes a state-transition relation. For example, the action $x' = x+1$ means approximately the same thing as the programming-language statement $x := x + 1$. However, the action is much simpler because it talks only about x and says nothing about another variable y, while the assignment statement may (or may not) assert that y doesn't change. TLA allows you to write specifications essentially the same way advocated in [82]. However, the specification becomes a single mathematical for-

mula. This opens up a whole new realm of possibilities. Among other things, it provides an elegant way to formalize and systematize all the reasoning used in concurrent system verification.

The moral of TLA is: if you're not writing a program, don't use a programming language. Programming languages are complicated and have many ugly properties because a program is input to a compiler that must generate reasonably efficient code. If you're describing an algorithm, not writing an actual program, you shouldn't burden yourselves with those complications and ugly properties. The toy concurrent programming languages with which computer scientists have traditionally described algorithms are not as bad as real programming languages, but they are still uglier and more complicated than they need to be. Such a toy program is no closer to a real C or Java program than is a TLA formula. And the TLA formula is a lot easier to deal with mathematically than is a toy program. (Everything I say about programming languages applies just as well to hardware description languages. However, hardware designers are generally more sensible than to try to use low-level hardware languages for higher-level system descriptions.) Had I only realized this 20 years ago!

The first major step in getting beyond traditional programming languages to describe concurrent algorithms was Misra and Chandy's Unity. Unity simply eliminated the control state, so you just had a single global state that you reasoned about with a single invariant. You can structure the invariant any way you want; you're not restricted by the particular programming constructs with which the algorithm is described. The next step was TLA, which eliminated the programming language and allowed you to write your algorithm directly in mathematics. This provides a much more powerful and flexible way of describing the next-state relation.

An amusing footnote to this paper is that, after reading an earlier draft, Simon Lam claimed that he deserved credit for the idea of describing actions as formulas with primed and unprimed variables. A similar notation for writing postconditions dates from the 70s, but that's not the same as actually specifying the action in this way. I had credited Rick Hehner's 1984 *CACM* article, but I figured there were probably earlier instances. After a modest amount of investigation, I found one earlier published use—in [50].

[103] **Decomposing Specifications of Concurrent Systems** (with Martín Abadi). *Programming Concepts, Methods and Calculi*, Ernst-Rüdiger

Olderog editor. (Proceedings of the IFIP TC2/WG2.1/WG2.2/WG2.3 Working Conference, Procomet '94, San Miniato, Italy.) North-Holland, (1994) 327–340.

See the discussion of [112].

[104] **Open Systems in TLA** (with Martín Abadi). *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing,* (August 1994) 81–90.

See the discussion of [112].

[105] **TLZ (Abstract)**. *Z User's Workshop, Cambridge 1994.* J.P. Bowen and J.A. Hall (Eds.) 267–268.

Z is a formal specification language that describes a system by writing actions—essentially the same kinds of actions that appear in a TLA specification. It was developed by Mike Spivey and others at Oxford for specifying sequential programs. When someone develops a method for sequential programs, they usually think that it will also handle concurrent programs—perhaps by adding an extra feature or two. I had heard that this wasn't true of the Z developers, and that they were smart enough to realize that Z did not handle concurrency. Moreover, Z is based on mathematics not programming languages, so it is a fairly nice language.

TLA assumes an underlying logic for writing actions. The next step was obvious: devise a language for specifying concurrent systems that extends Z with the operators of TLA. Equally obvious was the name of such a language: TLZ.

In the Spring of 1991, I visited Oxford and gave a talk on TLA, pointing out how naturally it could be combined with Z. The idea was as popular as bacon and eggs at Yeshiva University. Tony Hoare was at Oxford, and concurrency at Oxford meant CSP. The Z community was interested only in combining Z with CSP—which is about as natural as combining predicate logic with C++.

A couple of years later, I was invited to give a talk at the Z User's Meeting. I dusted off the TLZ idea and presented it at the meeting. Again, I encountered a resounding lack of interest.

Had TLA been adopted by the Z community, it might have become a lot more popular. On the other hand, not being attached to Z meant that I didn't have to live with Z's drawbacks and was free to design a more elegant language for specifying actions. The result was TLA$^+$, described in [127].

[106] **An Old-Fashioned Recipe for Real Time** (with Martín Abadi). *ACM Transactions on Programming Languages and Systems 16*, 5 (September 1994) 1543–1571. Also appeared as SRC Research Report 91. A preliminary version appeared in *Real-Time: Theory in Practice*, J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors (1992), Springer-Verlag, 1–27.

As explained in the discussion of [51], it's been clear for a long time that assertional methods for reasoning about concurrent algorithms can easily handle real-time algorithms. Time just becomes another variable. That hasn't stopped academics from inventing new formalisms for handling time. (Model checking real-time algorithms does raise new problems, since they are inherently not finite-state.) So, when de Roever held a workshop on formalisms for real-time systems, it was a good opportunity to show off how easily TLA deals with real-time algorithms. We also proved some new results about nonZeno specifications. I believe this paper introduced the terms *Zeno* and *nonZeno*, though the notion of Zeno behaviors had certainly occurred to others. It does seem to have been the first to observe the relation between nonZenoness and machine closure. Abadi has the following to say about this paper:

> For me, this paper was partly a vindication of some work I had done with [Gordon] Plotkin [**A Logical View of Composition**, *Theoretical Computer Science 114*, 1 (June 1993), 3–30], where we explored the definition and properties of the "while" operator ($\rightarrow\!\triangleright$). I believe that you thought that the work was a formal game, so I was pleased to find that we could use it in this paper.

The paper uses as an example a mutual exclusion protocol due to Michael Fischer. This example has an amusing history. When I was working on [73], I sent Fischer email describing my algorithm and asking if he knew of a better solution. He responded

> No, I don't, but the practical version of the problem sounds very similar to the problem of bus allocation in contention networks. I wonder if similar solutions couldn't be used? For example, how about...

He followed this with a simple, elegant protocol based on real-time delays. Because of its real-time assumptions, his protocol didn't solve

the problem that motivated [73]. I mentioned this algorithm in [73], but had forgotten about it by the time of de Roever's workshop. Fred Schneider reminded me of it when he used it as an example in his talk at the workshop. We then incorporated the example in our paper. I later mentioned this to Fischer, who had no memory of the protocol and even claimed that it wasn't his. Fortunately, I happen to have saved his original email, so I had the proof. The message, complete with message number, is cited in the paper—the only instance of such a citation that I know of.

[107] **Specifying and Verifying Fault-Tolerant Systems** (with Stephan Merz). *Formal Techniques in Real-Time and Fault-Tolerant Systems*, H. Langmaack, W.-P. de Roever, J. Vytopil editors. Lecture Notes in Computer Science, number 863, Springer-Verlag, (September 1994) 41–76.

Willem-Paul de Roever invited me to give a talk at this symposium. I was happy to have a podium to explain why verifying fault-tolerant, real-time systems should not be a new or especially difficult problem. This was already explained in [106] for real-time systems, but I knew that there would be people who thought that fault-tolerance made a difference. Moreover, de Roever assured me that Lübeck, where the symposium was held, is a beautiful town. (He was right.) So, I decided to redo in TLA the proof from [51]. However, when the time came to write the paper, I realized that I had overextended myself and needed help. The abstract states

> We formally specify a well known solution to the Byzantine generals problem and give a rigorous, hierarchically structured proof of its correctness. We demonstrate that this is an engineering exercise, requiring no new scientific ideas.

However, there were few computer scientists capable of doing this straightforward engineering exercise. Stephan Merz was one of them. So, I asked him to write the proof, which he did with his usual elegance and attention to detail. I think I provided most of the prose and the initial version of the TLA specification, which Merz modified a bit. The proof was all Merz's.

[108] **How to Write a Long Formula**. *FACJ 6*(5) (September/October 1994) 580–584. Also appeared as SRC Research Report 119.

Specifications often contain formulas that are a page or two long.

Mathematicians almost never write formulas that long, so they haven't developed the notations needed to cope with them. This article describes my notation for using indentation to eliminate parentheses in a formula consisting of nested conjunctions and disjunctions. I find this notation very useful when writing specifications. The notation is part of the formal syntax of TLA$^+$ (see [127]).

[109] **Introduction to TLA**. SRC Technical Note 1994-001 (December 1994).

This is a very brief (7-page) introduction to what TLA formulas mean.

[110] **Adding "Process Algebra" to TLA**. Unpublished (January 1995).

At the Dagstuhl workshop described in the discussion of [114], I was impressed by the elegance of the process-algebraic specification presented by Rob van Glabbeek. The ability to encode control state in the process structure permits one to express some specifications quite nicely in CCS. However, pure CCS forces you to encode the entire state in the process structure, which is impractical for real specifications. I had the idea of trying to get the best of both worlds by combining CCS and TLA, and wrote this preliminary note about it. I hoped to work on this with van Glabbeek but, although he was interested, he was busy with other things and we never discussed it, and I never did anything more with the idea. When I wrote this note, I wasn't sure if it was a good idea. I now think that examples in which adding CCS to TLA would significantly simplify the specification are unlikely to arise in practice. So, I don't see any reason to complicate TLA in this way. But, someone else may feel otherwise.

[111] **What Process Algebra Proofs Use Instead of Invariance**. Unpublished (January 1995).

Working on [110] got me thinking about how process-algebraic proofs work. This draft note describes my preliminary thoughts about what those proofs use instead of invariance. I never developed this far enough to know if it's right.

[112] **Conjoining Specifications** (with Martín Abadi). *ACM Transactions on Programming Languages and Systems 17*, 3 (May 1995), 507–534. Also appeared as SRC Research Report 118.

The obvious way to write an assume/guarantee specification is in the form E implies M, where E specifies what we assume of the environ-

ment and M specifies what the system guarantees. That is what we did in [97]. However, such a specification allows behaviors in which the system violates the guarantee and the environment later violates its assumption. This paper presents a better way to write the specification that we discovered later. Instead of E implies M, we take as the specification the stronger condition that M must remain true at least one step longer than E is. This enabled us to simplify and strengthen our results.

This paper contains two major theorems, one for decomposing closed-system specifications and another for composing open-system specifications. A preliminary conference version of the result for closed systems appeared in [103]. A preliminary conference version of the second appeared in [104].

Although the propositions and theorems in this paper are not in principle difficult, it was rather hard to get the details right. We couldn't have done it without writing careful, structured proofs. So, I wanted those proofs published. But rigorous structured proofs, in which all the details are worked out, are long and boring, and the referees didn't read them. Since the referees hadn't read the proofs, the editor didn't want to publish them. Instead, she wanted simply to publish the paper without proofs. I was appalled that she was willing to publish theorems whose proofs hadn't been checked, but was unwilling to publish the unchecked proofs. But, I sympathized with her reluctance to kill all those trees, so we agreed that she would find someone to referee the proof and we would publish the appendix electronically. The referee read the proofs carefully and found three minor errors, which were easily corrected. Two of the errors occurred when we made changes to one part of the proof without making corresponding changes to another. The third was a careless mistake in a low-level statement. When asked, the referee said that the hierarchical structure, with all the low-level details worked out, made the proofs quite clear and easy to check.

When I learned that ACM was going to publish some appendices in electronic form only, I was worried about their ability to maintain an electronic archive that would enable people to obtain an appendix twenty or fifty years later. Indeed, when I checked in August of 2011, none of the methods for obtaining a copy from the ACM that were printed with the article worked, and the appendix did not seem to be on their web site. It was still available from a Princeton University ftp site. (The link above is to a version of the paper containing the

appendix.)

[113] **TLA in Pictures**. *IEEE Transactions on Software Engineering SE-21*, 9 September 1995), 768–775. Also appeared as SRC Research Report 127.

Back in the 50s and 60s, programmers used flowcharts. Eventually, guided by people like Dijkstra and Hoare, we learned that pictures were a bad way to describe programs because they collapsed under the weight of complexity, producing an incomprehensible spaghetti of boxes and arrows. In the great tradition of learning from our mistakes how to make the same mistake again, many people decided that drawing pictures was a wonderful way to specify systems. So, they devised graphical specification languages.

Not wanting to be outdone, I wrote this paper to show that you can write TLA specifications by drawing pictures. It describes how to interpret as TLA formulas the typical circles and arrows with which people describe state transitions. These diagrams represent safety properties. I could also have added some baroque conventions for adding liveness properties to the pictures, but there's a limit to how silly I will get. When I wrote the paper, I actually did think that pictures might be useful for explaining parts of specifications. But I have yet to encounter any real example where they would have helped.

This paper contains, to my knowledge, the only incorrect "theorem" I have ever published. It illustrates that I can be as lazy as anyone in not bothering to check "obvious" assertions. I didn't published a correction because the theorem, which requires an additional hypothesis, was essentially a footnote and didn't affect the main point of the paper. Also, I was curious to see if anyone would notice the error. Apparently, no one did. I discovered the error in writing [115]

[114] **The RPC-Memory Specification Problem: Problem Statement** (with Manfred Broy). *Formal Systems Specification: The RPC-Memory Specification Case Study*, Manfred Broy, Stephan Merz, and Katharina Spies editors. Lecture Notes in Computer Science, number 1169, (1996), 1–4.

I don't remember how this came about, but Manfred Broy and I organized a Dagstuhl workshop on the specification and verification of concurrent systems. (I'm sure I agreed to this knowing that Broy and his associates would do all the real organizing.) We decided to pose a problem that all participants were expected to solve. This is the

problem statement.

There is an interesting footnote to this workshop. As explained in the discussion of [50], I don't believe in writing specifications as a conjunction of the properties that the system should satisfy. Several participants used this approach. I thought that the high-level specification was sufficiently trivial that by now, people would be able to specify it in this way. However, Reino Kurki-Suonio noticed an error that was present in all the "purely axiomatic" specifications—that is, ones that mentioned only the interface, without introducing internal variables.

[115] **A TLA Solution to the RPC-Memory Specification Problem** (with Martín Abadi and Stephan Merz). *Formal Systems Specification: The RPC-Memory Specification Case Study*, Manfred Broy, Stephan Merz, and Katharina Spies editors. Lecture Notes in Computer Science, number 1169, (1996), 21–66.

Since the problem posed in [114] was devised by both Broy and me, I felt it was reasonable for me to present a TLA solution. Martín Abadi, Stephan Merz, and I worked out a solution that Merz and I presented at the workshop. Afterwards, we worked some more on it and finally came up with a more elegant approach that is described in this paper. I believe that Abadi and I wrote most of the prose. Merz wrote the actual proofs, which he later checked using his embedding of TLA in Isabelle. We all contributed to the writing of the specifications.

This is the only example I've encountered in which the pictures of TLA formulas described in [113] were of some use. In fact, I discovered the error in [113] when I realized that one of the pictures in this paper provided a counterexample to its incorrect theorem.

[116] **How to Tell a Program from an Automobile**. In *A Dynamic and Quick Intellect*, John Tromp editor (1996)—a *Liber Amicorum* issued by the CWI in honor of Paul Vitanyi's 25-year jubilee.

I wrote this brief note in January, 1977. It came about because I was struck by the use of the term *program maintenance*, which conjured up in my mind images of lubricating the branch statements and cleaning the pointers. So, I wrote this to make the observation that programs are mathematical objects that can be analyzed logically. I was unprepared for the strong emotions this stirred up among my colleagues at Massachusetts Computer Associates, who objected vehemently to my thesis. So, I let the whole thing drop. Years later, when I was

invited to submit a short paper for the volume honoring Vitanyi, I decided that this paper would be appropriate because he had learned to drive only a few years earlier. The paper retains its relevance, since programmers still don't seem to understand the difference between a program and an automobile.

[117] **Refinement in State-Based Formalisms**. SRC Technical Note 1996-001 (December 1996).

A brief (7-page) note explaining what refinement and dummy variables are all about. It also sneaks in an introduction to TLA. In September 2004, Tommaso Bolognesi pointed out an error in the formula on the bottom of page 4 and suggested a correction. Instead of modifying the note, I've decided to leave the problem of finding and correcting the error as an exercise for the reader.

[118] **Marching to Many Distant Drummers** (with Tim Mann). Unpublished (May 1997).

In 1990, there were two competing proposals for a time service for the Internet. One was from the DEC networking group and the other was in an RFC by David Mills. The people in DEC asked me for theoretical support for their belief that their proposal was better than that of Mills. I asked Tim Mann to help me. We decided that we didn't like either proposal very much, and instead we wrote a note with our own idea for an algorithm to obtain the correct time in an Internet-like environment. We sat on the idea for a few years, and eventually Tim presented it at a Dagstuhl workshop on time synchronization. We then began writing a more rigorous paper on the subject. This is as far as we got. The paper is mostly finished, but it contains some minor errors in the statements of the theorems and the proofs are not completed. We are unlikely ever to work on this paper again.

[119] **Processes are in the Eye of the Beholder**. *Theoretical Computer Science, 179*, (1997), 333–351. Also appeared as SRC Research Report 132.

The notion of a process has permeated much of the work on concurrency. Back in the late 70s, I was struck by the fact that a uniprocessor computer could implement a multiprocess program, and that I had no idea how to prove the correctness of this implementation. Once I had realized that a system was specified simply as a set of sequences of states, the problem disappeared. Processes are just a particular way

of viewing the state, and different views of the same system can have different numbers of processors.

A nice example of this is an N-buffer producer/consumer system, which is usually viewed as consisting of a producer and a consumer process. But we can also view it as an N-process system, with each buffer being a process. Translating the views into concrete programs yields two programs that look quite different. It's not hard to demonstrate their equivalence with a lot of hand waving. With TLA, it's easy to replace the hand waving by a completely formal proof. This paper sketches how.

I suspected that it would be quite difficult and perhaps impossible to prove the equivalence of the two programs with process algebra. So, at the end of the paper, I wrote "it would be interesting to compare a process-algebraic proof ... with our TLA proof." As far as I know, no process algebraist has taken up the challenge. I figured that a proof similar to mine could be done in any trace-based method, such as I/O automata. But, I expected that trying to make it completely formal would be hard with other methods. Yuri Gurevich and Jim Huggins decided to tackle the problem using Gurevich's evolving algebra formalism (now called *abstract state machines*). The editor processing my paper told me that they had submitted their solution and suggested that their paper and mine be published in the same issue, and that I write some comments on their paper. I agreed, but said that I wanted to comment on the final version. I heard nothing more about their paper, so I assumed that it had been rejected. I was surprised to learn, three years later, that the Gurevich and Huggins paper, *Equivalence is in the Eye of the Beholder*, appeared right after mine in the same issue of *Theoretical Computer Science*. They chose to write a "human-oriented" proof rather than a formal one. Readers can judge for themselves how easy it would be to formalize their proof.

[120] **How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor**. *IEEE Transactions on Computers 46*, 7 (July 1997), 779–782. Also appeared as SRC Research Report 96.

This paper was inspired by Kourosh Gharachorloo's thesis. The problem he addressed was how to execute a multiprocess program on a computer whose memory did not provide sequential consistency (see [35]), but instead required explicit synchronization operations (such as Alpha's memory barrier instruction). He presented a method for deducing what synchronization operations had to be added to a pro-

gram. I realized that, if one proved the correctness of an algorithm using the two-arrow formalism of [33], the proof would tell you what synchronization operations were necessary. This paper explains how.

[121] **Substitution: Syntactic versus Semantic**. SRC Technical Note 1998-004 (March 1998). Rejected by *Information Processing Letters*.

What I find to be the one subtle and somewhat ugly part of TLA involves substitution in *Enabled* predicates. In the predicate *Enabled A*, there is an implicit quantification over the primed variables in *A*. Hence, mathematical substitution does not distribute over the *Enabled* operator. This four-page note explains that the same problem arises in most program logics because there is also an implicit quantification in the sequential-composition (semicolon) operator, so substitution does not distribute over semicolon. Apparently, no one had noticed this before because they hadn't tried using programming logics to do the sort of things that are easy to do in TLA.

[122] **The Part-Time Parliament**. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169. Also appeared as SRC Research Report 49. This paper was first submitted in 1990, setting a personal record for publication delay.

A fault-tolerant file system called *Echo* was built at SRC in the late 80s. The builders claimed that it would maintain consistency despite any number of non-Byzantine faults, and would make progress if any majority of the processors were working. As with most such systems, it was quite simple when nothing went wrong, but had a complicated algorithm for handling failures based on taking care of all the cases that the implementers could think of. I decided that what they were trying to do was impossible, and set out to prove it. Instead, I discovered the Paxos algorithm, described in this paper. At the heart of the algorithm is a three-phase consensus protocol. Dale Skeen seems to have been the first to have recognized the need for a three-phase protocol to avoid blocking in the presence of an arbitrary single failure. However, to my knowledge, Paxos contains the first three-phase commit algorithm that is a real algorithm, with a clearly stated correctness condition and a proof of correctness.

I thought, and still think, that Paxos is an important algorithm. Inspired by my success at popularizing the consensus problem by describing it with Byzantine generals, I decided to cast the algorithm in terms of a parliament on an ancient Greek island. Leo Guibas sug-

gested the name *Paxos* for the island. I gave the Greek legislators the names of computer scientists working in the field, transliterated with Guibas's help into a bogus Greek dialect. (Peter Ladkin suggested the title.) Writing about a lost civilization allowed me to eliminate uninteresting details and indicate generalizations by saying that some details of the parliamentary protocol had been lost. To carry the image further, I gave a few lectures in the persona of an Indiana-Jones-style archaeologist, replete with Stetson hat and hip flask.

My attempt at inserting some humor into the subject was a dismal failure. People who attended my lecture remembered Indiana Jones, but not the algorithm. People reading the paper apparently got so distracted by the Greek parable that they didn't understand the algorithm. Among the people I sent the paper to, and who claimed to have read it, were Nancy Lynch, Vassos Hadzilacos, and Phil Bernstein. A couple of months later I emailed them the following question:

> Can you implement a distributed database that can tolerate the failure of any number of its processes (possibly all of them) without losing consistency, and that will resume normal behavior when more than half the processes are again working properly?

None of them noticed any connection between this question and the Paxos algorithm.

I submitted the paper to *TOCS* in 1990. All three referees said that the paper was mildly interesting, though not very important, but that all the Paxos stuff had to be removed. I was quite annoyed at how humorless everyone working in the field seemed to be, so I did nothing with the paper. A number of years later, a couple of people at SRC needed algorithms for distributed systems they were building, and Paxos provided just what they needed. I gave them the paper to read and they had no problem with it. So, I thought that maybe the time had come to try publishing it again.

Meanwhile, the one exception in this dismal tale was Butler Lampson, who immediately understood the algorithm's significance. He mentioned it in lectures and in a paper, and he interested Nancy Lynch in it. De Prisco, Lynch, and Lampson published their version of a specification and proof. Their papers made it more obvious that it was time for me to publish my paper. So, I proposed to Ken Birman, who was then the editor of *TOCS*, that he publish it. He suggested revising it, perhaps adding a TLA specification of the algorithm. But

rereading the paper convinced me that the description and proof of the algorithm, while not what I would write today, was precise and rigorous enough. Admittedly, the paper needed revision to take into account the work that had been published in the intervening years. As a way of both carrying on the joke and saving myself work, I suggested that instead of my writing a revision, it be published as a recently rediscovered manuscript, with annotations by Keith Marzullo. Marzullo was willing, Birman agreed, and the paper finally appeared.

There was an amusing typesetting footnote to this. To set off Marzullo's annotations, I decided that they should be printed on a gray background. ACM had recently acquired some wonderful new typesetting software, and TOCS was not accepting camera-ready copy. Unfortunately, their wonderful new software could not do shading. So, I had to provide camera-ready copy for the shaded text. Moreover, their brilliant software could accept this copy only in floating figures, so Marzullo's annotations don't appear quite where they should. Furthermore, their undoubtedly expensive software wasn't up to typesetting serious math. (After all, it's a computing journal, so why should they have to typeset formulas?) Therefore, I had to provide the camera-ready copy for the definitions of the invariants in section A2, which they inserted as Figure 3 in the published version. So, the fonts in that figure don't match those in the rest of the paper.

[123] **Reduction in TLA** (with Ernie Cohen). *CONCUR'98 Concurrency Theory*, David Sangiorgi and Robert de Simone editors. Lecture Notes in Computer Science, number 1466, (1998), 317–331.

Reduction is a method of deducing properties of a system by reasoning about a coarser-grained model—that is, one having larger atomic actions. Reduction was probably first used informally for reasoning about multiprocess programs to justify using the coarsest model in which each atomic operation accesses only a single shared variable. The term *reduction* was coined by Richard Lipton, who published the first paper on the topic. Reduction results have traditionally been based on an argument that the reduced (coarser-grained) model is in some sense equivalent to the original. For terminating programs that simply produce a result, equivalence just means producing the same result. But for reactive programs, it has been difficult to pin down exactly what equivalence means. TLA allowed me for the first time to understand the precise relation between the original and the reduced systems. In [95], I proved a result for safety specifications that

generalized the standard reduction theorems. This result formulated reduction as a temporal theorem relating the original and reduced specifications—that is, as a property of individual behaviors. This formulation made it straightforward to extend the result to handle liveness, but I didn't get around to working on the extension until late in 1996.

Meanwhile, Ernie Cohen had been working on reduction using Kleene algebra, obtaining elegant proofs of nice, general results for safety properties. I showed him the TLA version and my preliminary results on liveness, and we decided to collaborate. This paper is the result. We translated his more general results for safety into TLA and obtained new results for liveness properties. The paper promises a complete proof and a more general result in a later paper. The result exists, but the later paper is unlikely ever to appear. A draft of the complete proof is available from the web version of this document.

[124] **Composition: A Way to Make Proofs Harder**. *Compositionality: The Significant Difference (Proceedings of the COMPOS'97 Symposium)*, Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli editors. Lecture Notes in Computer Science, number 1536, (1998), 402–423.

Systems are complicated. We master their complexity by building them from simpler components. This suggests that to master the complexity of reasoning about systems, we should prove properties of the separate components and then combine those properties to deduce properties of the entire system. In concurrent systems, the obvious choice of component is the process. So, compositional reasoning has come to mean deducing properties of a system from properties of its processes.

I have long felt that this whole approach is rather silly. You don't design a mutual exclusion algorithm by first designing the individual processes and then hoping that putting them together guarantees mutual exclusion. Similarly, anyone who has tried to deduce mutual exclusion from properties proved by considering the processes in isolation knows that it's the wrong way to approach the problem. You prove mutual exclusion by finding a global invariant, and then showing that each process's actions maintains the invariant. TLA makes the entire reasoning process completely mathematical—the specifications about which one reasons are mathematical formulas, and proving correctness means proving a single mathematical formula. A mathe-

matical proof is essentially decompositional: you apply a deduction rule to reduce the problem of proving a formula to that of proving one or more simpler formulas.

This paper explains why traditional compositional reasoning is just one particular, highly constrained way of decomposing the proof. In most cases, it's not a very natural way and results in extra work. This extra work is justified if it can be done by a computer. In particular, decomposition along processes makes sense if the individual processes are simple enough to be verified by model checking. TLA is particularly good for doing this because, as illustrated by [119], it allows a great deal of flexibility in choosing what constitutes a process.

[125] **Proving Possibility Properties**. *Theoretical Computer Science 206*, 1–2, (October 1998), 341–352. Also appeared as SRC Research Report 137.

One never wants to assert possibility properties as correctness properties of a system. It's not interesting to know that a system *might* produce the correct answer. You want to know that it will never produce the wrong answer (safety) and that it eventually will produce an answer (liveness). Typically, possibility properties are used in branching-time logics that cannot express liveness. If you can't express the liveness property that the system must do something, you can at least show that the system might do it. In particular, process algebras typically can express safety but not liveness. But the trivial system that does nothing implements any safety property, so process algebraists usually rule out such trivial implementations by requiring bisimulation—meaning that the implementation allows all the same possible behaviors as the specification.

People sometimes argue that possibility properties are important by using the ambiguities of natural language to try to turn a liveness property into a possibility property. For example, they may say that it should be possible for the user of a bank's ATM to withdraw money from his account. However, upon closer examination, you don't just want this to be possible. (It's possible for me to withdraw money from an ATM, even without having an account, if a medium-sized meteorite hits it.) The real condition is that, if the user presses the right sequence of buttons, then he must receive the money.

Since there is no reason to prove possibility properties of a system, I was surprised to learn from Bob Kurshan—a very reasonable person—that he regularly uses his model checker to verify possibility properties.

Talking to him, I realized that although verifying possibility properties tells you nothing interesting about a system, it can tell you something interesting about a specification, which is a mathematical model of the system. For example, you don't need to specify that a user can hit a button on the ATM, because you're specifying the ATM, not the user. However, we don't reason about a user interacting with the ATM; we reason about a mathematical model of the user and the ATM. If, in that mathematical model, it were impossible for the button to be pushed, then the model would be wrong. Proving possibility properties can provide sanity checks on the specification. So, I wrote this paper explaining how you can use TLA to prove possibility properties of a specification—even though a linear-time temporal logic like TLA cannot express the notion of possibility.

I originally submitted this paper to a different journal. However, the editor insisted that, to publish the paper, I had to add a discussion about the merits of branching-time versus linear-time logic. I strongly believe that it's the job of an editor to judge the paper that the author wrote, not to get him to write the paper that the editor wants him to. So, I appealed to the editor-in-chief. After receiving no reply for several months, I withdrew the paper and submitted it to *TCS*.

[126] **A Lazy Caching Proof in TLA** (with Peter Ladkin, Bryan Olivier, and Denis Roegel). *Distributed Computing 12*, 2/3, (1999), 151–174.

At some gathering (I believe it was the workshop where I presented [103]), Rob Gerth told me that he was compiling a collection of proofs of the lazy caching algorithm of Afek, Brown, and Merritt. I decided that this was a good opportunity to demonstrate the virtues of TLA, so there should be a TLA solution. In particular, I wanted to show that the proof is a straightforward engineering task, not requiring any new theory. I wanted to write a completely formal, highly detailed structured proof, but I didn't want to do all that dull work myself. So, I enlisted Ladkin, Olivier (who was then a graduate student of Paul Vitanyi in Amsterdam), and Roegel (who was then a graduate student of Dominique Mery in Nancy), and divided the task among them. However, writing a specification and proof is a process of continual refinement until everything fits together right. Getting this done in a long-distance collaboration is not easy, and we got tired of the whole business before a complete proof was written. However, we had done enough to write this paper, which contains specifications and a high-level overview of the proof.

[127] **Specifying Concurrent Systems with TLA$^+$**. *Calculational System Design*. M. Broy and R. Steinbrüggen, editors. IOS Press, Amsterdam, (1999), 183–247.

I was invited to lecture at the 1998 Marktoberdorf summer school. One reason I accepted was that I was in the process of writing a book on concurrency, and I could use the early chapters of the book as my lecture notes. However, I decided to put aside (perhaps forever) that book and instead write a book on TLA$^+$. I was able to recycle much material from my original notes for the purpose. For the official volume of published notes for the course, I decided to provide this, which is a preliminary draft of the first several chapters of [144].

[128] **TLA$^+$ Verification of Cache-Coherence Protocols** (with Homayoon Akhiani et al.). Rejected from *Formal Methods '99* (February 1999).

Mark Tuttle, Yuan Yu, and I formed a small group applying TLA to verification problems at Compaq. Our two major projects, in which we have had other collaborators, have been verifications of protocols for two multiprocessor Alpha architectures. We thought it would be a good idea to write a paper describing our experience doing verification in industry. The FM'99 conference had an Industrial Applications track, to include "Experience reports [that] might describe a case study or industrial project where a formal method was applied in practice." So, we wrote this paper and submitted it. It was rejected. One of the referees wrote, "The paper is rather an experience report than a scientific paper." Our paper is indeed short on details, since neither system had been released at that time and almost all information about it was still company confidential. However, I think it still is worth reading if you're interested in what goes on in the industrial world.

[129] **Should Your Specification Language Be Typed?** (with Larry Paulson). *ACM Transactions on Programming Languages and Systems 21*, 3 (May 1999) 502-526. Also appeared as SRC Research Report 147.

In 1995, I wrote a diatribe titled *Types Considered Harmful*. It argued that, although types are good for programming languages, they are a bad way to formalize mathematics. This implies that they are bad for specification and verification, which should be mathematics rather than programming. My note apparently provoked some discus-

sion, mostly disagreeing with it. I thought it might be fun to promote a wider discussion by publishing it, and *TOPLAS* was, for me, the obvious place. Andrew Appel, the editor-in-chief at the time, was favorably disposed to the idea, so I submitted it. Some of my arguments were not terribly sound, since I know almost nothing about type theory. The referees were suitably harsh, but Appel felt it would still be a good idea to publish a revised version along with rebuttals. I suggested that it would be better if I and the referees cooperated on a single balanced article presenting both sides of the issue. The two referees agreed to shed their anonymity and participate. Larry Paulson was one of the referees. It soon became apparent that Paulson and I could not work with the other referee, who was rabidly pro-types. (At one point, he likened his situation to someone being asked by a neo-Nazi to put his name on a "balanced" paper on racism.) So, Paulson and I wrote the paper by ourselves. We expected that the other referee would write a rebuttal, but he apparently never did.

[130] **Model Checking TLA$^+$ Specifications** (with Yuan Yu and Panagiotis Manolios). In *Correct Hardware Design and Verification Methods (CHARME '99)*, Laurence Pierre and Thomas Kropf editors. Lecture Notes in Computer Science, number 1703, Springer-Verlag, (September 1999) 54–66.

Despite my warning him that it would be impossible, Yuan Yu wrote a model checker for TLA$^+$ specifications. He succeeded beyond my most optimistic hopes. This paper is a preliminary report on the model checker. I was an author, at Yu's insistence, because I gave him some advice on the design of the model checker (more useful advice than just don't do it). Manolios worked at SRC as a summer intern and contributed the state-compression algorithm that is described in the paper, but which ultimately was not used in the model checker.

[131] **How (La)TEX changed the face of Mathematics**. *Mitteilungen der Deutschen Mathematiker-Vereinigung 1/2000* (Jan 2000) 49–51.

Günther Ziegler interviewed me by email for this note, which delves into the history of LaTeX.

[132] **Fairness and Hyperfairness**. *Distributed Computing 13*, 4 (2000), 239–245. Also appeared as SRC Research Report 152. A preliminary version of this paper was rejected by *Concur* 99.

In 1993, Attie, Francez, and Grumberg published a paper titled *Fair-*

ness and Hyperfairness in Multi-Party Interactions. This was a follow-up to the 1988 paper *Appraising Fairness in Languages for Distributed Programming* by Apt, Francez, and Katz, which attempted to define fairness. I have long believed that the only sensible formal definition of fairness is machine closure, which is essentially one of the conditions mentioned by Apt, Francez, and Katz. (They called it *feasibility* and phrased it as a condition on a language rather than on an individual specification.) I refereed the Attie, Francez, and Grumberg paper and found it rather uninteresting because it seemed to be completely language-dependent. They apparently belonged to the school, popular in the late 70s and early 80s, that equated concurrency with the CSP programming constructs. I wrote a rather unkind review of that paper, which obviously didn't prevent its publication. Years later, it suddenly struck me that there was a language-independent definition of hyperfairness—or more precisely, a language-independent notion that seemed to coincide with their definition on a certain class of CSP programs. I published this paper for three reasons: to explain the new definition of hyperfairness; to explain once again that fairness is machine closure and put to rest the other two fairness criteria conditions of Apt, Francez, and Katz; and, in some small way, to make up for my unkind review of the Attie, Francez, and Grumberg paper.

[133] **Archival References to Web Pages**. Ninth International World Wide Web Conference: Poster Proceedings (May 2000), page 74..

On several occasions, I've had to refer to a web page in a published article. The problem is that articles remain on library shelves for many decades, while URLs are notoriously transitory. This short note describes a little trick of mine for referring to a web page by something more permanent than a URL. You can discover the trick by looking at the description on page 2 of how to find the web version of this document. Although my idea is ridiculously simple and can be used by anyone right now, I've had a hard time convincing anyone to use it. (Because it's only a pretty good solution and not perfect, people prefer to do nothing and wait for the ideal solution that is perpetually just around the corner.) Since I was going to be in the neighborhood, I decided to try to publicize my trick with this poster at WWW9. Some people I spoke to there thought it was a nice idea, but I'm not optimistic that anyone will actually use it.

Unfortunately, the version posted by the conference is missing a gif file.

[134] **Disk Paxos** (with Eli Gafni). *Distributed Computing 16*, 1 (2003) 1–20.

In 1998, Jim Reuter of DEC's storage group asked me for a leader-election algorithm for a network of processors and disks that they were designing. The new wrinkle to the problem was that they wanted a system with only two processors to continue to operate if either processor failed. We could assume that the system had at least three disks, so the idea was to find an algorithm that achieved fault tolerance by replicating disks rather than processors. I convinced them that they didn't want a leader-election protocol, but rather a distributed state-machine implementation (see [27]). At the time, Eli Gafni was on sabbatical from UCLA and was consulting at SRC. Together, we came up with the algorithm described in this paper, which is a disk-based version of the Paxos algorithm of [122].

Gafni devised the initial version of the algorithm, which didn't look much like Paxos. As we worked out the details, it evolved into its current form. Gafni wanted a paper on the algorithm to follow the path with which the algorithm had been developed, starting from his basic idea and deriving the final version by a series of transformations. We wrote the first version of the paper in this way. However, when trying to make it rigorous, I found that the transformation steps weren't as simple as they had appeared. I found the resulting paper unsatisfactory, but we submitted it anyway to PODC'99, where it was rejected. Gafni was then willing to let me do it my way, and I turned the paper into its current form.

A couple of years after the paper was published, Mauro J. Jaskelioff encoded the proof in Isabelle/HOL and mechanically checked it. He found about a dozen small errors. Since I have been proposing Disk Paxos as a test example for mechanical verification of concurrent algorithms, I have decided not to update the paper to correct the errors he found. Anyone who writes a rigorous mechanically-checked proof will find them.

[135] **Disk Paxos (Conference Version)** (with Eli Gafni). *Distributed Computing: 14th International Conference, DISC 2000*, Maurice Herlihy, editor. Lecture Notes in Computer Science number 1914, Springer-Verlag, (2000) 330–344.

This is the abridged conference version of [134].

[136] **When Does a Correct Mutual Exclusion Algorithm Guar-**

antee Mutual Exclusion (with Sharon Perl and William Weihl). *Information Processing Letters 76*, 3 (March 2000), 131–134.

Mutual exclusion is usually defined to mean that two processes are not in their critical section *at the same time*. Something Dan Scales said during a conversation made me suddenly realize that conventional mutual exclusion algorithms do not satisfy that property. I then conjectured how that property could be satisfied, and Perl and Weihl proved that my conjecture was correct. This paper explains why mutual exclusion had not previously been achieved, and how to achieve it—all in less than five pages.

[137] **Lower Bounds on Consensus**. unpublished note (March 2000).

This short note is described by its abstract:

> We derive lower bounds on the number of messages and the number of message delays required by a nonblocking fault-tolerant consensus algorithm, and we show that variants of the Paxos algorithm achieve those bounds.

I sent it to Idit Keidar who told me that the bounds I derived were already known, so I forgot about it. About a year later, she mentioned that she had cited the note in:

> Idit Keidar and Sergio Rajsbaum. On the Cost of Fault-Tolerant Consensus When There Are No Faults - A Tutorial. MIT Technical Report MIT-LCS-TR-821, May 24 2001. Preliminary version in SIGACT News 32(2), Distributed Computing column, pages 45-63, June 2001 (published in May 15th).

I then asked why they had cited my note if the results were already known. She replied,

> There are a few results in the literature that are similar, but not identical, because they consider slightly different models or problems. This is a source of confusion for many people. Sergio and I wrote this tutorial in order to pull the different known results together. Hopefully, it can help clarify things up.

[138] **The Wildfire Challenge Problem** (with Madhu Sharma, Mark Tuttle, and Yuan Yu). Rejected from CAV 2001 (January 2001).

From late fall 1996 through early summer 1997, Mark Tuttle, Yuan

Yu, and I worked on the specification and verification of the cache-coherence protocol for a computer code-named Wildfire. We worked closely with Madhu Sharma, one of Wildfire's designers. We wrote a detailed specification of the protocol as well as a specification of the memory model that it was supposed to implement. We then proved various properties, but did not attempt a complete proof. In early 2000, Madhu, Mark, and I wrote a specification of a higher-level abstract version of the protocol.

There was one detail of the protocol that struck me as particularly subtle. I had the idea of publishing an incorrect version of the specification with that detail omitted as a challenge problem for the verification community. I did that and put it on the Web in June, 2000. To further disseminate the problem, we wrote this description of it for the CAV (Computer Aided Verification) conference.

[139] **Paxos Made Simple**. *ACM SIGACT News (Distributed Computing Column) 32*, 4 (Whole Number 121, December 2001) 51–58.

At the PODC 2001 conference, I got tired of everyone saying how difficult it was to understand the Paxos algorithm, published in [122]. Although people got so hung up in the pseudo-Greek names that they found the paper hard to understand, the algorithm itself is very simple. So, I cornered a couple of people at the conference and explained the algorithm to them orally, with no paper. When I got home, I wrote down the explanation as a short note, which I later revised based on comments from Fred Schneider and Butler Lampson. The current version is 13 pages long, and contains no formula more complicated than $n_2 > n_1$.

[140] **Specifying and Verifying Systems with TLA$^+$** (with John Matthews, Mark Tuttle, and Yuan Yu). Proceedings of the Tenth ACM SIGOPS European Workshop (2002), 45–48.

This describes our experience at DEC/Compaq using TLA$^+$ and the TLC model checker on several systems, mainly cache-coherence protocols. It is shorter than, and more up-to-date than [128].

[141] **Arbiter-Free Synchronization**. *Distributed Computing 16*, 2/3, (2003) 219–237.

With the bakery algorithm of [12], I discovered that mutual exclusion, and hence all conventional synchronization problems, could be solved with simple read/write registers. However, as recounted in the

description of [22], such a register requires an arbiter. This leads to the question: what synchronization problems can be solved without an arbiter? Early on, I devised a more primitive kind of shared register that can be implemented without an arbiter, and I figured out how to solve the producer/consumer problem with such registers. I think that hardware designers working on self-timed circuits probably already knew that producer/consumer synchronization could be implemented without an arbiter. (If not, they must have figured it out at about the same time I did.) Hardware people used Muller C-elements instead of my shared registers, but it would have been obvious to them what I was doing.

In Petri nets, arbitration appears explicitly as conflict. A class of Petri nets called marked graphs, which were studied in the early 70s by Anatol Holt and Fred Commoner, are the largest class of Petri nets that are syntactically conflict-free. Marked-graph synchronization is a natural generalization of producer/consumer synchronization. It was clear to me that marked-graph synchronization can be implemented without an arbiter, though I never bothered writing down the precise algorithm. I assumed that marked graphs describe precisely the class of synchronization problems that could be solved without an arbiter.

That marked-graph synchronization can be implemented without an arbiter is undoubtedly obvious to people like Anatol Holt and Chuck Seitz, who are familiar with multiprocess synchronization, Petri nets, and the arbiter problem. However, such people are a dying breed. So, I thought I should write up this result before it was lost. I had been procrastinating on this for years when I was invited to submit an article for a special issue of *Distributed Computing* celebrating the 20th anniversary of the PODC conference. The editors wanted me to pontificate for a few pages on the past and future of distributed computing—something I had no desire to do. However, it occurred to me that it would be fitting to contribute some unpublished 25-year-old work. So, I decided to write about arbiter-free synchronization.

Writing the paper required me to figure out the precise arbiter-free implementation of marked graphs, which wasn't hard. It also required me to prove my assumption that marked graphs were all one could implement without an arbiter. When I tried, I realized that my assumption was wrong. There are multiprocess synchronization problems not describable by marked graphs that can be solved without an arbiter. The problem was more complicated than I had realized.

I wish I knew exactly what can be done without an arbiter, but

I don't. It turns out that I don't really understand arbiter-free synchronization. Lack of understanding leads to messy exposition. I understand the results about the equivalence of registers, and I have nice, crisp ways of formalizing and proving these results. But the other results in the paper are a mess. They're complicated and I don't have a good way of formalizing them. Had I written this twenty-five years ago, I would probably have kept working on the problem before publishing anything. But I don't have the time I once did for mulling over hard problems. I decided it was better to publish the results I have, even though they're messy, and hope that someone else will figure out how to do a better job.

[142] **A Discussion With Leslie Lamport**. An interview in *IEEE Distributed Systems Online 3*, 8 .

In the spring of 2002, Dejan Milojicic proposed interviewing me for an IEEE on-line publication. He volunteered to send me the questions in advance, and to send me the transcript afterwards for correction. This seemed pretty silly, so I just wrote my answers. The "interview" was conducted by a few email exchanges.

[143] **Lower Bounds for Asynchronous Consensus**. *Future Directions in Distributed Computing*, André Schiper, Alex A. Shvartsman, Hakim Weatherspoon, and Ben Y. Zhao, editors. Lecture Notes in Computer Science number 2584, Springer, (2003) 22–23.

The FuDiCo (Future Directions in Distributed Computing) workshop was held in a lovely converted monastery outside Bologna. I was supposed to talk about future directions, but restrained my natural inclination to pontificate and instead presented some new lower-bound results. The organizers wanted to produce a volume telling the world about the future of distributed computing research, so everyone was supposed to write a five-page summary of their presentations. I used only two pages. Since I didn't have rigorous proofs of my results, and I expected to discover special-case exceptions, I called them approximate theorems. This paper promises that future papers will give precise statements and proofs of the theorems, and algorithms showing that the bounds are tight. Despite my almost perfect record of never writing promised future papers, I actually wrote up the case of non-Byzantine failures in [153]. I intend some day to write another paper with the general results for the Byzantine case. Really.

[144] **Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers**. *Addison-Wesley (2002).*

The complete book of TLA$^+$. The first seven chapters (83 pages) are a rewritten version of [127]. That and the chapter on the TLC model checker are about as much of the book as I expect people to read. The web page contains errata and some exercises and examples.

[145] **Checking Cache-Coherence Protocols with TLA$^+$** (with Rajeev Joshi, John Matthews, Serdar Tasiran, Mark Tuttle, and Yuan Yu). *Formal Methods in System Design 22*, 2 (March 2003) 125-131.

Yet another report on the TLA$^+$ verification activity at Compaq. It mentions some work that's been done since we wrote [140].

[146] **High-Level Specifications: Lessons from Industry** (with Brannon Batson). *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors. Lecture Notes in Computer Science number 2852, Springer, (2003) 242–262.

I was invited to speak about TLA at the FMCO symposium. I didn't feel that I had anything new to say, so I asked Brannon Batson who was then at Intel to help me prepare the talk and the paper. Brannon is a hardware designer who started using TLA$^+$ while at Compaq and has continued using it at Intel. The most interesting part of this paper is Section 4, which is mainly devoted to Brannon's description of how his group is using TLA$^+$ in their design process.

Section 5 was inspired by the symposium's call for papers, whose list of topics included such fashionable buzzwords as "object-oriented", "component-based", and "information hiding". It explains why those concepts are either irrelevant to or are a bad idea for high-level specification.

[147] **The Future of Computing: Logic or Biology**. Text of a talk given at Christian Albrechts University, Kiel on 11 July 2003.

I was invited to give a talk to a general university audience at Kiel. Since I'm not used to giving this kind of non-technical talk, I wrote it out in advance and read it. Afterwards, I received several requests for a copy to be posted on the Web. So, here it is.

[148] **Consensus on Transaction Commit** (with Jim Gray). *ACM Transactions on Database Systems 31*, 1 (2006), 133-160. Also appeared

as Microsoft Research Technical Report MSR-TR-2003-96 (February 2005).

In [143], I announced some lower-bound results for the consensus problem. One result states that two message delays are required to choose a value, and a relatively large number of processors are needed to achieve that bound. When writing a careful proof of this result, I realized that it required the hypothesis that values proposed by two different processors could be chosen in two message delays. This led me to realize that fewer processors were needed if there were only one processor whose proposed value could be chosen in two message delays, and values proposed by other processors took longer to be chosen. In fact, a simple modification to the Paxos algorithm of [122] accomplished this.

I then looked for applications of consensus in which there is a single special proposer whose proposed value needs to be chosen quickly. I realized there is a "killer app"—namely, distributed transaction commit. Instead of regarding transaction commit as one consensus problem that chooses the single value *commit* or *abort*, it could be presented as a set of separate consensus problems, each choosing the commit/abort desire of a single participant. Each participant then becomes the special proposer for one of the consensus problems. This led to what I call the Paxos Commit algorithm. It is a fault-tolerant (non-blocking) commit algorithm that I believed had fewer message delays in the normal (failure-free) case than any previous algorithm. I later learned that an algorithm published by Guerraoui, Larrea, and Schiper in 1996 had the same normal-case behavior.

Several months later, Jim Gray and I got together to try to understand the relation between Paxos and the traditional Two-Phase Commit protocol. After a couple of hours of head scratching, we figured out that Two-Phase Commit is the trivial version of Paxos Commit that tolerates zero faults. That realization and several months of procrastination led to this paper, which describes the Two-Phase Commit and Paxos Commit algorithms and compares them. It also includes an appendix with TLA$^+$ specifications of the transaction-commit problem and of the two algorithms.

[149] **On Hair Color in France** (with Ellen Gilkerson). *Annals of Improbable Research*, Jan/Feb 2004, 18–19.

While traveling in France, Gilkerson and I observed many blonde

women, but almost no blonde men. Suspecting that we had stumbled upon a remarkable scientific discovery, we endured several weeks of hardship visiting the auberges and restaurants of France to gather data. After several years of analysis and rigorous procrastination, we wrote this paper. Much of our magnificent prose was ruthlessly eliminated by the editor to leave space for less important research.

[150] **Formal Specification of a Web Services Protocol** (with James E. Johnson, David E. Langworthy, and Friedrich H. Vogt). *Electronic Notes in Theoretical Computer Science 105*, M. Bravetti and G. Zavattaro editors. (December 2004) 147–158.

Fritz Vogt spent part of a sabbatical at our lab during the summer and fall of 2003. I was interested in getting TLA$^+$ used in the product groups at Microsoft, and Fritz was looking for an interesting project involving distributed protocols. Through his contacts, we got together with Jim Johnson and Dave Langworthy, who work on Web protocols at Microsoft in Redmond. Jim and Dave were interested in the idea of formally specifying protocols, and Jim suggested that we look at the Web Services Atomic Transaction protocol as a simple example. Fritz and I spent part of our time for a couple of months writing it, with a lot of help from Jim and Dave in understanding the protocol. This paper describes the specification and our experience writing it. The specification itself is at `http://research.microsoft.com/users/lamport/tla/ws-at.html`.

This was a routine exercise for me, as it would have been for anyone with a moderate amount of experience specifying concurrent systems. Using TLA$^+$ for the first time was a learning experience for Fritz. It was a brand new world for Jim and Dave, who had never been exposed to formal methods before. They were happy with the results. Dave began writing specifications by himself, and has become something of a TLA$^+$ guru for the Microsoft networking group. We submitted this paper to WS-FM 2004 as a way of introducing the Web services community to formal methods and TLA$^+$.

[151] **Cheap Paxos** (with Mike Massa). *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2004)* held in Florence in June-July 2004.

A system that can tolerate a single non-Byzantine failure requires three processors. It has long been realized that only two of those processors need to maintain the system state, but the third processor

must take part in every decision to maintain consistency. Mike Massa, at the time an engineer at Microsoft, observed that if we weaken the fault-tolerance guarantee, then the third processor needs to be used only in the case of failure or repair of one of the other two processors. The third processor can then be a less powerful machine or a process run occasionally on a computer devoted to other tasks. I generalized his idea to a variation of the Paxos algorithm of [122] called Cheap Paxos that tolerates up to f failures with $f + 1$ main processors and f auxiliary ones. A paper on this algorithm was rejected from the PODC and DISC conferences. Most of the referees thought that it just presented the old idea that only the main processors need to maintain the system state, not realizing that it differed from the old approach because the remaining f processors need not take part in every decision.

One review contained a silly assertion that it was easy to solve the problem in a certain way. When trying to prove his or her assertion false, I discovered a somewhat simpler version of Cheap Paxos that achieved the same result as the original version. (The new algorithm wasn't at all what the referee said should be done, which was impossible.) This paper describes the simpler algorithm. The original algorithm actually has some advantage over the new one, but it remains unpublished.

[152] **Implementing and Combining Specifications**. Unpublished note (September 2004).

I wrote this note to help some Microsoft developers understand how they could write TLA$^+$ specifications of the software they were designing. Their biggest problem was figuring out how to specify an API (Application Programming Interface) in TLA$^+$, since there were no published examples of such specifications. The note also explains two other things they didn't understand: what it means to implement an API specification and how to use an API specification in specifying a system that calls the API.

[153] **Lower Bounds for Asynchronous Consensus**. *Distributed Computing 19*, 2 (2006), 79–103. Also appeared as Microsoft Research Technical Report MSR-TR-2004-72 (July 2004, revised August 2005).

This paper contains the precise statements and proofs of the results announced in [143] for the non-Byzantine case. It also includes another result showing that a completely general consensus algorithm

cannot be faster than the Paxos algorithm of [122] in the presence of conflicting requests. However, there are two exceptional cases in which this result does not hold, and the paper presents potentially useful optimal algorithms for both cases.

[154] **Generalized Consensus and Paxos**. Microsoft Research Technical Report MSR-TR-2005-33 (15 March 2005).

In [153], I proved lower bounds for the number of message delays required to reach consensus. I showed that the best algorithms can reach consensus in the normal case in 2 message delays. This result in turn led me to a new version of the Paxos algorithm of [122] called Fast Paxos, described in [158], that achieves this bound. However, Fast Paxos can take 3 message delays in the event of conflict, when two values are proposed concurrently. I showed in [153] that this was unavoidable in a general algorithm, so this seemed to be the last word.

It then occurred to me that, in the state-machine approach (introduced in [27]), such conflicting proposals arise because two different commands are issued concurrently by two clients, and both are proposed as command number i. This conflict is necessary only if the two proposed commands do not commute. If they do, then there is no need to order them. This led me to a new kind of agreement problem that requires dynamically changing agreement on a growing partially ordered set of commands. I realized that generalizing from partially ordered sets of commands to a new mathematical structure I call a *c-struct* leads to a generalized consensus problem that covers both ordinary consensus and this new dynamic agreement problem. I also realized that Fast Paxos can be generalized to solve this new problem. I wrote up these results in March 2004. However, I was in the embarrassing position of having written a paper generalizing Fast Paxos without having written a paper about Fast Paxos. So, I just let the paper sit on my disk.

I was invited to give a keynote address at the 2004 DSN conference, and I decided to talk about fast and generalized Paxos. Fernando Pedone came up after my talk and introduced himself. He said that he and André Schiper had already published a paper with the same generalization from the command sequences of the state-machine approach to partially ordered sets of commands, together with an algorithm that achieved the same optimal number of message delays in the absence of conflict. It turns out that their algorithm is different from the generalized Paxos algorithm. There are cases in which generalized

Paxos takes only 2 message delays while their algorithm takes 3. But the difference in efficiency between the two algorithms is insignificant. The important difference is that generalized Paxos is more elegant.

I've been sitting on this paper for so long because it doesn't seem right to publish a paper on a generalization of Fast Paxos before publishing something about Fast Paxos itself. Since generalized Paxos is a generalization, this paper also explains Fast Paxos. But people's minds don't work that way. They need to understand Fast Paxos before they can really understand its generalization. So, I figured I would turn this paper into the second part of a long paper or monograph whose first part explains Fast Paxos. However, in recent years I've been discovering new Paxonian results faster than I can write them up. It therefore seems silly not to release a paper that I've already written about one of those results. So, I added a brief discussion of the Pedone-Schiper result and a citation to [153] and am posting the paper here. Now that I have written the Fast Paxos paper and submitted it for publication, I may rewrite this paper as part two of that one.

[155] **Real Time is Really Simple**. Microsoft Research Technical Report MSR-TR-2005-30 (4 March 2005). Rejected by *Formal Methods in Systems Design*.

It should be quite obvious that no special logic or language is needed to write or reason about real-time specifications. There's a simple way to do it: just use a variable to represent time. Martín Abadi and I showed in [106] that this can be done very elegantly in TLA. A simpler, more straightforward approach works with any sensible formal method, but it's too simple and obvious to publish. So instead, hundreds of papers and theses have been written about special real-time logics and languages—even though, for most purposes, there's no reason to use them instead of the simple, obvious approach. And since no one writes papers about the simple way of handling real time, people seem to assume that they need to use a real-time logic. Naturally, I find this rather annoying. So when I heard that a computer scientist was planning to write a book about one of these real-time logics, I decided it was time to write another paper explaining the simple approach.

Since you can't publish a new paper about an old idea, no matter how good the idea may be, I needed to find something new to add. The TLC model checker provided the opportunity I needed. The method described in [106] and [144] is good for specifying and reasoning about

real-time systems, but it produces specifications that TLC can't handle. TLC works only with the simpler approach, so I had an excuse for a new paper.

There's a naive approach for checking real-time specifications with TLC that I had thought for a while about trying. It involves checking the specification for all runs up to some maximum time value that one hopes is large enough to find any bugs. So I did that using as examples two versions of Fischer's mutual exclusion protocol, which is mentioned in the discussion of [106].

One possible reason to use a special real-time approach is for model checking. I figured that model checkers using special algorithms for real time should do much better than this naive approach, so I wanted some justification for using TLA$^+$ and TLC instead. Looking through the literature, I found that all the real-time model checkers seemed to use low-level languages that could describe only simple controllers. So I added the example of a distributed algorithm that they couldn't represent. Then I discovered that, since the papers describing it had been published, the Uppaal model checker had been enhanced with new language features that enabled it to model this algorithm. This left me no choice but to compare TLC with Uppaal on the example.

I asked Kim Larsen of Aalborg University, the developer of Uppaal, for help writing an Uppaal spec of the algorithm. Although I really did this because I'm lazy, I could justify my request because I had never used Uppaal and couldn't see how to write a nice model with it. Larsen got his colleague Arne Skou to write a model that was quite nice, though it did require a bit of a "hack" to encode the high-level constructs of TLA$^+$ in Uppaal's lower-level language. Skou was helped by Larsen and his colleague, Gerd Behrmann. As I expected, Uppaal was much faster than TLC—except in one puzzling case in which Uppaal ran out of memory.

I put the paper aside for a while. When I got back to it, I realized that there's a simple way of using TLC to do complete checking of these real-time specifications that is much faster than what I had been doing. The idea is so simple that I figured it was well known, and I kicked myself for not seeing it right away. I checked with Tom Henzinger, who informed me that the method was known, but it had apparently not been published. It seems to be an idea that is obvious to the experts and unknown to others. So this provided another incentive for publishing my paper, with a new section on how to use an explicit-time model checker like TLC to check real-time specs. Hen-

zinger also corrected a basic misunderstanding I had about real-time model checkers. Rather than trying to be faster, most of them try to be better by using continuous time. He wrote:

> If you are happy with discrete time, I doubt you can do any better [than my naive approach]. Uppaal, Kronos etc. deal with real-numbered time, and therefore rely on elaborate and expensive clock region constructions.

I was then inspired to do some more serious data gathering. I discovered the explanation of that puzzling case: Uppaal runs out of memory when the ratio of two parameters becomes too large. The results reported in the paper show that neither TLC nor Uppaal comes out clearly better on this example.

The Uppaal distribution comes with a model of a version of Fischer's algorithm, and I decided to get some data for that example too. Uppaal did clearly better than TLC on it. However, I suspected that the reason was not because real-time model checkers are better, but because TLC is less efficient for this kind of simple algorithm than a model checker that uses a lower-level language. So I got data for two ordinary model checkers that use lower-level languages, Spin and SMV. I was again lazy and got the developers of those model checkers, Gerard Holzmann and Ken McMillan, to do all the work of writing and checking the models.

I submitted this paper to the journal *Formal Methods in Systems Design*. I thought that the part about model checking was interesting enough to be worth putting into a separate conference paper. I therefore wrote [157], which was accepted at the 2005 Charme conference. However, the journal submission was rejected because it didn't contain enough new ideas.

[156] **How Fast Can Eventual Synchrony Lead to Consensus?** (with Partha Dutta and Rachid Guerraoui). *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2005)*.

During a visit I made to the EPFL in March 2004, Dutta and Guerraoui explained a problem they were working on. Asynchronous consensus algorithms like Paxos [122] maintain safety despite asynchrony, but are guaranteed to make progress only when the system becomes synchronous—meaning that messages are delivered in a bounded length of time. Dutta and Guerraoui were looking for an algorithm that always reaches agreement within a constant number of message delays

after the system becomes synchronous. This is a hard problem only if messages sent before the system becomes synchronous can be delivered arbitrarily far in the future. I took the solution they had come up with and combined it with Paxos to obtain the algorithm described is this paper. It's a nice solution to a mildly interesting theoretical problem with no apparent practical application. As I recall, I wanted to include a sentence in the paper saying this, but my co-authors sensibly pointed out that doing so would ensure the paper's rejection. (My co-authors don't remember this.) Computer scientists in this field must keep up the pretense that everything they do is practical.

[157] **Real-Time Model Checking is Really Simple**. *Correct Hardware Design and Verification Methods* (CHARME 2005), Dominique Borrione and Wolfgang J. Paul editors, Springer-Verlag Lecture Notes in Computer Science Volume 3725 (2005), 162–175.

This is an abridged version of [155], containing only the material on model checking.

[158] **Fast Paxos**. *Distributed Computing 19*, 2 (October 2006) 79–103. Also appeared as Microsoft Research Technical Report MSR-TR-2005-112 (14 July 2005). .

The Paxos consensus algorithm of [122] requires two message delays between when the leader proposes a value and when other processes learn that the value has been chosen. Since inventing Paxos, I had thought that this was the optimal message delay. However, sometime in late 2001 I realized that in most systems that use consensus, values aren't picked out of the air by the system itself; instead, they come from clients. When one counts the message from the client, Paxos requires three message delays. This led me to wonder whether consensus in two message delays, including the client's message, was in fact possible. I proved the lower-bound result announced in [143] that an algorithm that can make progress despite f faults and can achieve consensus in two message delays despite e faults requires more than $2e + f$ processes. The proof of that result led me pretty quickly to the Fast Paxos algorithm described here. Fast Paxos generalizes the classic Paxos consensus algorithm. It can switch between learning in two or three message delays depending on how many processes are working. More precisely, it can achieve learning in two message delays only in the absence of concurrent conflicting proposals, which [153] shows is the best a general algorithm can do.

[159] **Measuring Celebrity**. *Annals of Improbable Research*, Jan/Feb 2006, 14–15.

In September 2005, I had dinner with Andreas Podelski, who was visiting Microsoft's Cambridge Research Laboratory. He mentioned that his home page was the fourth item returned by a Google search on his first name. His casual remark inspired the scientific research reported here.

[160] **Checking a Multithreaded Algorithm with +CAL**. In *Distributed Computing: 20th International Conference, DISC 2006*, Shlomi Dolev, editor. Springer-Verlag (2006) 11–163.

Yuan Yu told me about a multithreaded algorithm that was later reported to have a bug. I thought that writing the algorithm in PlusCal (formerly called +CAL) [161] and checking it with the TLC model checker [127] would be a good test of the PlusCal language. This is the story of what I did. The PlusCal specification of the algorithm and the error trace it found are available on the web.

[161] **The PlusCal Algorithm Language**. *Theoretical Aspects of Computing–ICTAC 2009*, Martin Leucker and Carroll Morgan editors. Lecture Notes in Computer Science, number 5684, 36–60.

PlusCal (formerly called +CAL) is an algorithm language. It is meant to replace pseudo-code for writing high-level descriptions of algorithms. An algorithm written in PlusCal is translated into a TLA+ specification that can be checked with the TLC model checker [127]. This paper describes the language and the rationale for its design. A language manual and further information are available on the Web.

An earlier version was rejected from POPL 2007. Based on the reviews I received and comments from Simon Peyton-Jones, I revised the paper and submitted it to TOPLAS, but it was again rejected. It may be possible to write a paper about PlusCal that would be considered publishable by the programming-language community. However, such a paper is not the one I want to write. For example, two of the three TOPLAS reviewers wanted the paper to contain a formal semantics—something that I would expect people interested in using PlusCal to find quite boring. (A formal TLA+ specification of the semantics is available on the Web.) I therefore decided to publish it as an invited paper in the ICTAC conference proceedings.

[162] **TLA$^+$**. Chapter in *Software Specification Methods: An Overview Using a Case Study*, Henri Habrias and Marc Frappier, editors. Hermes, April 2006.

I was asked to write a chapter for this book, which consists of a collection of formal specifications of the same example system written in a multitude of different formalisms. The system is so simple that the specification should be trivial in any sensible formalism. I bothered writing the chapter because it seemed like a good idea to have TLA$^+$ represented in the book, and because it wasn't much work since I was able to copy a lot from the Z specification in Jonathan Bowen's chapter and simply explain how and why the Z and TLA$^+$ specifications differ. Bowen's chapter is available at `http://www.jpbowen.com/pub/ssm-z2.pdf`.

Because the example is so simple and involves no concurrency, its TLA$^+$ specification is neither interesting nor enlightening. However, my comments about the specification process may be of some interest.

[163] **Implementing Dataflow With Threads**. *Distributed Computing 21*, 3 (2008), 163–181. Also appeared as Microsoft Research Technical Report MSR-TR-2006-181 (December 2006)..

In the summer of 2005, I was writing an algorithm in PlusCal [161] and essentially needed barrier synchronization as a primitive. The easiest way to do this in PlusCal was to write a little barrier synchronization algorithm. I used the simplest algorithm I could think of, in which each process maintains a single 3-valued variable—the *Barrier*1 algorithm of this paper. The algorithm seemed quite nice, and I wondered if it was new. A Web search revealed that it was. (In 2008, Wim Hesselink informed me that he had discovered this algorithm in 2001, but he had "published" it only in course notes.) I was curious about what barrier synchronization algorithm was used inside the Windows operating system and how it compared with mine, so I asked Neill Clift. He and John Rector found that my algorithm outperformed the one inside Windows. Meanwhile, I showed my algorithm to Dahlia Malkhi, who suggested some variants, including the paper's *Barrier*2 algorithm.

By around 1980, I knew that the producer/consumer algorithm introduced in [23] should generalize to an arbitrary marked graph, but I never thought it important enough to bother working out the details. (Marked graphs, which specify dataflow computation, are described in

the discussion of [141].) I realized that these new barrier synchronization algorithms should also be instances of that generalization. The fact that the barrier algorithms worked well on a real multiprocessor made the general algorithm seem more interesting. Further thought revealed that the good performance of these barrier algorithms was not an accident. They have optimal caching behavior, and that optimal behavior can be achieved in the general case. All this makes the general synchronization algorithm relevant for the coming generation of multicore processor chips.

[164] **Leslie Lamport: The Specification Language TLA$^+$**. In *Logics of Specification Languages*, Dines Bjørner and Martin C. Henson, editors. Springer (2008), 616–620.

This is a "review" of a chapter by Stephan Merz in the same book. It is mainly a brief account of the history behind TLA and TLA$^+$. It includes an interesting quote from Brannon Battson. (See [146].)

[165] **Computation and State Machines**. Unpublished (February 2008).

I have long thought that computer science is about concepts, not languages. On a visit to the University of Lugano in 2006, the question arose of what that implied about how computer science should be taught. This is a first, tentative attempt at an answer.

[166] **The Mailbox Problem** (with Marcos Aguilera and Eli Gafni). *Distributed Computing 23*, 2 (2010), 113–134. (A shorter version appeared in *Proceedings of the 22nd International Symposium on Distributed Computing, (DISC 2008)*, 1–15.).

This paper addresses a little synchronization problem that I first thought about in the 1980s. When Gafni visited MSR Silicon valley in 2008, I proposed it to him and we began working on it. I thought the problem was unsolvable, but we began to suspect that there was a solution. Gafni had an idea for an algorithm, but instead of trying to understand the idea, I asked for an actual algorithm. We then went through a series of iterations in which Gafni would propose an algorithm, I'd code it in PlusCal (see [161]) and let the model checker find an error trace, which I would then give to him. (At some point, he learned enough PlusCal to do the coding himself, but he never installed the TLA+ tools and I continued to run the model checker.) This process stopped when Aguilera joined MSR and began collaborating with us. He turned Gafni's idea into an algorithm that the model checker ap-

proved of. Gafni and Aguilera came up with the impossibility results. Aguilera and I did most of the actual writing, which included working out the details of the proofs.

[167] **Teaching Concurrency**. *ACM SIGACT News Volume 40*, Issue 1 (March 2009), 58–62.

Idit Keidar invited me to submit a note to a distributed computing column in SIGACT News devoted to teaching concurrency. In an introduction, she wrote that my note "takes a step back from the details of where, what, and how, and makes a case for the high level goal of teaching students how to think clearly." What does it say about the state of computer science education that one must make a case for teaching how to think clearly?

[168] **Vertical Paxos and Primary-Backup Replication** (with Dahlia Malkhi and Lidong Zhou). *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009*, Srikanta Tirthapura and Lorenzo Alvisi, editors. ACM (2009), 312–313.

This paper came out of much discussion between Malkhi, Zhou, and myself about reconfiguration. Some day, what we did may result in a long paper about state-machine reconfiguration containing these results and others that have not yet been published. The ideas here are related to the original, unpublished version of [151].

[169] **Computer Science and State Machines**. *Concurrency, Compositionality, and Correctness (Essays in Honor of Willem-Paul de Roever).* Dennis Dams, Ulrich Hannemann, and Martin Steffen editors. Lecture Notes in Computer Science, number 5930 (2010), 60–65.

This is the six-page version of [165]. I think it is also the first place I have mentioned the Whorfian syndrome in print. It is structured around a lovely simple example in which an important hardware protocol is derived from a trivial specification by substituting an expression for the specification's variable. This example is supporting evidence for the thesis of [167] that computation should be described with mathematics. (Substitution of an expression for a variable is an elementary operation of mathematics, but is meaningless in a programming language.)

[170] **Reconfiguring a State Machine** (with Dahlia Malkhi and Lidong

Zhou). *ACM SIGACT News Volume 41*, Issue 1 (March 2010)..

This paper describes several methods of reconfiguring a state machine. All but one of them can be fairly easily derived from the basic state-machine reconfiguration method presented in the Paxos paper [122]. We felt that it was worthwhile publishing them because few people seemed to understand the basic method. (The basic method has a parameter α that was I took to be 3 in [122] because I stupidly thought that everyone would realize that the 3 could be any positive integer.) The one new algorithm, here called the "brick wall" method, is just sketched. It is described in detail in [171].

This paper was rejected by the 2008 PODC conference. Idit Keidar invited us to submit it as a tutorial to her distributed computing column in SIGACT News.

[171] **Stoppable Paxos** (with Dahlia Malkhi and Lidong Zhou). Unpublished (April 2009).

This paper contains a complete description and proof of the "brick wall" algorithm that was sketched in [170]. It was rejected from the 2008 DISC conference.

[172] **Byzantizing Paxos by Refinement**. *Distributed Computing: 25th International Symposium: DISC 2011*, David Peleg, editor. Springer-Verlag (2011) 211–224.

The Castro-Liskov algorithm (Miguel Castro and Barbara Liskov, *Practical Byzantine Fault Tolerance and Proactive Recovery*, TOCS 20:4 [2002] 398–461) intuitively seems like a modification of Paxos [122] to handle Byzantine failures, using $3n + 1$ processes instead of $2n + 1$ to handle n failures. In 2003 I realized that a nice way to think about the algorithm is that $2n + 1$ non-faulty processes are trying to implement ordinary Paxos in the presence of n malicious processes—each good process not knowing which of the other processes are malicious. Although I mentioned the idea in lectures, I didn't work out the details.

The development of TLAPS, the TLA$^+$ proof system, inspired me to write formal TLA$^+$ specifications of the two algorithms and a TLAPS-checked proof that the Castro-Liskov algorithm refines ordinary Paxos. This paper describes the results. The complete specifications and proof are available at `http://research.microsoft.com/users/lamport/tla/byzpaxos.html`.

[173] **Leaderless Byzantine Paxos**. *Distributed Computing: 25th International Symposium: DISC 2011*, David Peleg, editor. Springer-Verlag (2011) 141–142.

This two-page note describes a simple idea that I had in 2005. I have found the Castro-Liskov algorithm and other "Byzantine Paxos" algorithms unsatisfactory because they use a leader and, for progress, they require detecting and removing a malicious leader. My idea was to eliminate the leader by using a synchronous Byzantine agreement algorithm to implement a virtual leader. The note is too short to discuss the practical details, but they seem to be straightforward.