A Retrospective of

Proving the Correctness of Multiprocess Programs

Leslie Lamport — Researcher Emeritus, Microsoft

13 November 2024

I began writing multiprocess algorithms in 1973. I quickly learned how easy it was to get them wrong, so it was important to prove that they were correct. I happened to learn about Robert Floyd's 1967 paper on proving correctness properties of terminating sequential (single-process) programs [6]. It inspired my 1977 paper *Proving the Correctness of Multiprocess Programs*.

Floyd proved two kinds of properties:

- *Partial correctness*, which asserts that if a formula called the *precondition* is true when the program is started, then the program can terminate only in a state satisfying a formula called the *postcondition*.
- *Termination*, which asserts that the program must eventually terminate.

Few multiprocess algorithms were supposed to terminate, so I had to generalize these two classes of properties. My generalizations, as I now describe them, were:

- *Safety*, which asserts what the program is allowed to do.
- *Liveness*, which asserts what the program must eventually do.

Only later were these informal definitions made rigorous—safety by me in 1984 [1], and liveness by Bowen Alpern and Fred Schneider in 1985 [2]. A correctness property was defined to be a predicate that is true or false of an individual execution, and Alpern and Schneider showed that every correctness property is the conjunction of a safety property and a liveness property. In retrospect, I find it remarkable that my intuition led me to generalize partial correctness to these two new concepts that turned out to be fundamental.[1]

[1] I took the names *safety* and *liveness* from Petri nets, where safety is a particular safety property and liveness is not a correctness property.

Introducing the concepts of safety and liveness was a small part of the paper. However, it was the part that most influenced software engineers. The rest of my paper is about how to prove these properties.

The paper describes how to prove a kind of safety property called invariance, which asserts that a formula, called an *invariant*, is true in all states of all possible executions of the program.[2] The paper asserts that any safety property can be expressed as an invariant by adding extra variables to the program—a fact that became obvious with the rigorous definition of safety. For example, if we add to a program a Boolean variable *pre* that is initially true if and only if the precondition is true, and whose value never changes, then partial correctness is invariance of the formula asserting that if *pre* is true and the program has terminated, then the postcondition is true.

Floyd used flowcharts to describe programs, pointing out that a program written in any programming language could be represented by a flowchart. A proof of partial correctness consisted of an annotation of the flowchart with a formula P_a for each arc a of the flowchart, where P_a equals the precondition if a is an input arc and equals the postcondition if a is an output arc, and a proof that, if control is at an input arc a of any flowchart box, then executing that box moves control to an arc b for which P_b is true.

What Floyd never stated, but was obvious to me, was that if we let π be a new variable whose value equals (the name of) the arc at which control is, then what a Floyd proof shows is that the conjunction of the formulas $(\pi = a) \Rightarrow P_a$ is an invariant of the program, where $\Rightarrow$ denotes implication. (It is assumed that π always equals a program arc.) This formula is what is now called an *inductive* invariant, meaning that if a step of the program is executed in any possible state satisfying the invariant—even a state that can't be reached in any execution of the program—then the resulting step satisfies the invariant.

This observation led to the obvious generalization in my paper. A multiprocess program consists of a flowchart for each process, with a variable π_k for each process k whose value indicates the arc of that process's flowchart at which control is. An annotation that attaches formulas to each arc asserts the invariance of the formula consisting of the conjunction of all formulas $(\pi_k = a) \Rightarrow P_a$ for every arc a of every process k. (Again, there is the assumption that π_k always equals an arc of process k.) To prove that this formula is an (inductive) invariant, we must show that executing any flowchart box starting with the invariant true leaves the invariant true. To prove that

[2]The term *Invariant* has been given other meanings. I believe this definition from the paper was used by most writers of multiprocess algorithms.

the program satisfies an invariant *Inv*, we must find an annotation whose (inductive) invariant implies *Inv*.

Introducing multiple processes adds a complication to the method. With one process, proving invariance requires showing that executing a flowchart box with control on an input arc a and with P_a true leaves control on an output arc b with P_b true. With multiple processes, the proof must also show that if control at any other process is on an arc c with P_c true, then the execution leaves P_c true. This additional condition is now called *interference freedom*.

Multiple processes require an additional change. With Floyd's method, a formula P_a contains only the program's variables, not the added variable π. With multiple processes, proving even very simple invariance properties requires that any formula P_a of the annotation be able to contain the variables π_k for processes k other than the one containing arc a. For example, this is necessary to prove that a program containing two processes, each of which just increments a variable x by 1 and stops, increments x by 2.

A few other methods for proving invariance properties of multiprocess programs were also published around the same time [3, 8, 12]. They all represented programs differently, making the methods look superficially different. However, they were essentially equivalent. For the examples being considered, it made no practical difference which one was used. They were all largely ignored except for the method of Susan Owicki and David Gries [12], which I will call O-G. To understand why other methods were ignored, we must return to proofs of partial correctness of single-process programs.

Floyd's work inspired a 1969 paper by C.A.R. (Tony) Hoare that described an axiomatic method for proving partial correctness based on formulas of the form $\Pi\{S\}\Phi$, which asserts that program statement S satisfies partial correctness with precondition Π and postcondition Φ. While a proof in Hoare's logic is mathematically equivalent to a proof by Floyd's method, it has the nice property that the proof is structured in a way that mirrors the structure provided by the programming language. In particular, the proof of partial correctness of a statement can be broken into separate, independent proofs of partial correctness of its component statements. Because of this, Hoare's method became the standard way of writing proofs of partial correctness. Floyd's paper is often cited for inspiring Hoare's logic, but its method of proving partial correctness has been largely forgotten.

O-G was generally adopted as the way to prove invariance properties of multiprocess programs because it annotated programs written in a (simple) conventional programming language rather than flowcharts, and it was pre-

sented as a generalization of Hoare's logic rather than of Floyd's method. However, while Hoare's logic allows decomposing the proof of a statement into independent proofs of its components, O-G does not have this property. The need to prove interference freedom meant the condition to be satisfied by a statement of one process had to depend on the annotation of other processes. So, O-G offered no practical advantage over the other methods.

O-G had a problem not shared by the other methods. The formulas in an annotation have to refer to the control state. However, with the abolition of *goto* statements [4], programming languages provided no way to describe the control state. O-G requires adding auxiliary variables to capture the information about the control state needed in the proof. Those variables are today called history variables because they record information from previous states of the program.

Having to add history variables posed no practical difficulty. However, because O-G was presented without mentioning control state, it was impossible to write the invariant described by a program annotation. I disliked that because I realized that what a program does at any point in its execution depends only on its current state, not on what happened in the past. The program does the correct thing because it's in a correct state, which means because it satisfies an inductive invariant. Understanding why an algorithm is correct requires knowing what that inductive invariant is. It doesn't depend on the values of history variables, but it does depend on the control state. The values of the variables π_k are just as much a part of the program's state as the values of the program variables.

Not being able to describe the inductive invariant also meant that O-G lacked the simple mathematical explanation based on the invariant. I observed that most computer scientists did not understand that an O-G annotation describes an inductive invariant, and hence did not fully understand the method, until almost ten years after it was published.[3] About ten years after my paper was published, I observed that it began to be cited along with the O-G papers. Perhaps those two observations were related. In any event, O-G is now well understood and seems to be the basis for most current methods of proving invariance properties of multiprocess programs.

Floyd's paper also presents a method for proving termination, and my paper generalizes it to a method for proving some liveness properties. While I believe I understood the general strategy of proving those properties, I found my method to be complicated and ugly, and I was dissatisfied with

[3]For example, the inductive invariant is not mentioned in Edsger Dijkstra's 1976 explanation of O-G, which doesn't mention the control state [5].

it. In 1977, Amir Pnueli introduced temporal logic to computer science [14]. Susan Owicki and I realized that his temporal logic provided a better way to formalize proofs of liveness, and it could be combined with any method for proving invariance properties [13].

Like all such papers from that time, mine talks about proving properties of programs. But I wasn't interested in programs; I was interested in algorithms. I would write my algorithms in what looked like a programming language because that was how it was done. However, I came to realize that an algorithm is not a program. It is an abstract description of a computation, which can be implemented in many different programming languages.

Since an algorithm isn't a program, there is no reason it needs to be written in a programming language; and there is a good reason why it shouldn't be. Making it possible to write large programs that can be executed efficiently adds complications to programming languages that are unnecessary for describing algorithms. I realized that an algorithm can be viewed more abstractly as a state machine consisting of (1) a set of states, where a state is an assignment of values to variables, (2) a set of correct initial states, (3) a next-state relation describing the set of possible next states from any state, and (4) a temporal logic formula describing what steps must eventually be taken. For most algorithms, the set of possible states is infinite, because restricting them to a finite set of states would introduce distracting complications.[4] The inductive invariant was simply a formula containing the state machine's variables.

A state machine is easily described mathematically, without anything like a programming language. I eventually invented TLA, a simple extension to Pnueli's original temporal logic in which a state machine can be described as a single formula [10]. That formula has the form $\mathit{Init} \wedge \Box[\mathit{Next}]_v \wedge L$, where *Init* describes the initial states, *Next* describes the next-state relation, v lists the variables that make up the state, and L expresses the liveness requirement. The formulas *Init* and *Next* are ordinary math, with no temporal logic. All the complexity is in *Next*, which can be hundreds of lines long. Formula L is at most a few lines of temporal logic that describe what steps must be taken in terms of fairness assumptions about steps allowed by *Next*.

Although I was originally motivated by multiprocess algorithms, I realized that the way to avoid concurrency errors in a multiprocess program

[4]Most people don't realize that infinite sets are used for simplicity. We teach arithmetic to children with an infinite set of integers because making it a finite set would be more complicated.

was by reasoning about a higher-level abstraction of the program—that is, by reasoning about the algorithm the program was supposed to implement rather than about the program. A multiprocess program should implement an algorithm that correctly synchronizes the processes—but it's not usually called an algorithm because it's useful just for that program, so I call it an abstraction. Other computer scientists also realized the value of describing multiprocess programs more abstractly, and various methods were proposed for writing the abstractions. My experience writing and proving the correctness of multiprocess algorithms led me to believe that the methods that would work in practice could be viewed as describing a program as a state machine. Since their purpose was to help write correct programs, it seemed natural to many computer scientists that the state machine should be described in a language that resembled a programming language.

To avoid synchronization errors in multiprocess programs, we need to reason about the simplest, most abstract useful representation of how the processes interact. I have found that thinking in terms of programming languages limits one's ability to abstract. Mathematics is based on abstraction, and learning to think mathematically teaches how to write simpler, higher-level abstractions. I therefore now advocate using mathematics instead of programming languages for writing higher-level views of programs. There is also another reason to describe state machines mathematically.

What it means for a multiprocess algorithm or an abstract view of a multiprocess program to be correct can be subtle. When it is, I have found the best way to state what it means for a state machine to be correct is with a higher-level, more abstract state machine. This requires defining what it means for one state machine to implement another. Hoare observed in 1972 that showing a program T satisfies the partial correctness condition $\Pi\{S\}\Phi$ of a program S means proving $\overline{\Pi}\{T\}\overline{\Phi}$, where $\overline{\Pi}$ and $\overline{\Phi}$ are obtained from Π and Φ by substituting expressions containing the variables of T for the variables of S [7]. However, he had no way to prove that program T implements program S because one can't substitute expressions for variables in a program. Substituting $y + z$ for x in a precondition $x > 0$ yields the precondition $y + z > 0$. But substituting $y + z$ for x in a program's assignment statement $x := x + 1$ yields the meaningless statement $y + z := y + z + 1$. However, substitution of an expression for a variable is a normal operation in mathematics. In TLA, showing that the state machine described by formula T implements the state machine described by formula S means proving $T \Rightarrow \overline{S}$, where $\overline{S}$ is the formula obtained by substituting suitable expressions in the variables of T for the variables of S.

In the early 1990s I developed the language TLA$^+$ for writing TLA for-

mulas. Tools for checking the correctness of state machines written in TLA$^+$ have been implemented, including a proof checker as well as two model checkers that are quite effective at finding errors [9].

My 1977 paper led ultimately to TLA$^+$. If it has significantly influenced software engineers other than by introducing the concepts of safety and liveness, then it is through TLA$^+$. The Internet has made the world dependent on distributed multiprocess programs that must be reliable. Many of the software engineers who build multiprocess programs realize the need to write high-level abstract descriptions of those programs. There are a number of instances of successful industrial use of TLA$^+$ for that purpose [11]. However, I have been told that while software engineers are accustomed to using programming languages, few of them are comfortable with mathematics, and quite a few are actually afraid of it. Even some TLA$^+$ users want the language to be less mathematical and more like a programming language. As I write this, it is unclear how influential TLA$^+$, and thus my 1977 paper, will ultimately be.

Acknowledgment

Fred Schneider corrected some lapses in my memory and suggested other improvements to this retrospective.

References

[1] Mack W. Alford, Jean-Pierre Ansart, Günter Hommel, Leslie Lamport, Barbara Liskov, Geoff P. Mullery, and Fred B. Schneider, editors. *Distributed Systems: Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985, Munich, Germany*, volume 190 of *Lecture Notes in Computer Science*. Springer, 1985.

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

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

[4] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968.

[5] Edsger W. Dijkstra. A personal summary of the Gries-Owicki theory. In Edsger W. Dijkstra, editor, *Selected Writings on Computing: A Personal Perspective*, chapter EWD554, pages 188–199. Springer-Verlag, New York, Heidelberg, Berlin, 1982.

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

[7] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[8] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, July 1976.

[9] Leslie Lamport. TLA—temporal logic of actions. A web page at `https://lamport.azurewebsites.net/tla/tla.html`.

[10] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[11] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, April 2015.

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

[13] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

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