Teaching Concurrency

Leslie Lamport

26 January 2009
minor correction: 30 November 2009

I am not an academic. I have never even taken a computer science course. However, I have worked with a number of computer engineers (both hardware and software engineers), and I have seen what they knew and what they didn't know that I felt they should have. So, I have some thoughts about how concurrent computing should be taught. I am not concerned with traditional questions of curriculum—what facts should be stuffed into the student's brain. I long ago forgot most of the facts that I learned in school. What I have used throughout my career are the ways of thinking I learned when I was young, by some learning process that I have never understood. I can't claim to know the best way to teach computer engineers how to cope with concurrency. I do know that what they seem to be learning now is not helping them very much. I believe that what I am proposing here is worth a try.

I expect that the first question most computer scientists would ask about the teaching of concurrency is, what programming language should be used? This reflects the widespread syndrome in computer science of concentrating on language rather than substance. The modern field of concurrency started with Dijkstra's 1965 paper on the mutual exclusion problem [1]. For most of the 1970s, one "solved" the mutual exclusion problem by using semaphores or monitors or conditional critical regions or some other language construct. This is like solving the sorting problem by using a programming language with a *sort* command. Most of your colleagues can explain how to implement mutual exclusion using a semaphore. How many of them can answer the following question: Can one implement mutual exclusion without using lower-level constructs that, like semaphores, assume mutually exclusive access to a resource? Quite a few people who think they are experts on concurrency can't.

Teaching about concurrency requires teaching some very basic things

about computer science that are not now taught. The basic subject of computer science is computation. One would expect a computer science education to answer the question, what is a computation? See how many of your colleagues can give a coherent answer to that question. While there is no single answer that is appropriate in all contexts, there is one that is most useful to engineers: A computation is a sequence of steps.

The obvious next question is, what's a step? Western languages are verb oriented. Every sentence, even one asserting that nothing happens, contains a verb. At least to westerners, the obvious answer is that a step is the acting out of a verb. An *add* step performs an addition; a *send* step sends a message. However, the obvious answer is not the most useful one. Floyd and Hoare taught us to think of a computation as a sequence of states [2, 3]. A step is then a transition from one state to the next. It is more useful to think about states than sequences of steps because what a computing device does next depends on its current state, not on what steps it took in the past. Computer engineers should learn to think about a computation in terms of states rather than verbs.

I have found that a major problem confronting engineers when designing a system is understanding what the system is supposed to do. They are seldom faced with well-understood tasks like sorting. Most often they begin with only a vague idea of what the system should do. All too often they start implementing at that point. A problem must be understood before it can be solved. The great contribution of Dijkstra's paper on mutual exclusion was not his solution; it was stating the problem. (It is remarkable that, in this first paper on the subject, Dijkstra stated all the requirements that distinguish mutual exclusion from fundamentally simpler and less interesting problems.) Programming and hardware-design languages don't help an engineer understand what problem a system should solve. Thinking of computations as sequences of states, rather than as something described by a language, is the first step towards such understanding.

How should we describe computations? Most computer scientists would probably interpret this question to mean, what language should we use? Imagine an art historian answering "how would you describe impressionist painting?" by saying "in French". To describe a computation we need to describe a sequence of states. More often, we need to describe the set of computations that can be produced by some particular computing device, such as an algorithm. There is one method that works in practice: describing a set of computations by (1) the set of all initial initial states and (2) a next-state relation that describes, for every state, the possible next states—that is, the set of states reachable from that state by a single step. The

languages used by computer engineers describe computations in this way, but how many engineers or computer scientists understand this?

Once an engineer understands what a computation is and how it is described, she can understand the most important concept in concurrency: invariance. A computing device does the correct thing only because it maintains a correct state. Correctness of the state is expressed by an invariant—a predicate that is true in every state of every computation. We prove that a predicate is an (inductive) invariant by showing that it is true in every initial state, and that the next-state relation implies that if it is true in any state then it remains true in the next state. This method of reasoning, often called the inductive assertion method, was introduced by Floyd and Hoare. However, they expressed the invariant as a program annotation; most people were distracted by the language and largely ignored the essential idea behind the method.

Invariance is the key to understanding concurrent systems, but few engineers or computer scientists have learned to think in terms of invariants. Here is a simple example. Consider N processes numbered from 0 through $N-1$ in which each process i executes

$x[i] := 1;$
$y[i] := x[(i-1) \bmod N]$

and stops, where each $x[i]$ initially equals 0. (The reads and writes of each $x[i]$ are assumed to be atomic.) This algorithm satisfies the following property: after every process has stopped, $y[i]$ equals 1 for at least one process i. It is easy to see that the algorithm satisfies this property; the last process i to write $y[i]$ must set it to 1. But that process doesn't set $y[i]$ to 1 because it was the last process to write y. What a process does depends only on the current state, not on what processes wrote before it. The algorithm satisfies this property because it maintains an inductive invariant. Do you know what that invariant is? If not, then you do not completely understand why the algorithm satisfies this property. How can a computer engineer design a correct concurrent system without understanding it? And how can she understand it if she has not learned how to understand even a simple concurrent algorithm?

To describe a set of computations, we must describe its initial states and its next-state relation. How do we describe them? Ultimately, a description must be written in some language. An art historian must decide whether to describe impressionism in French or another language. I used some simple programming notation and English to describe the algorithm in the preceding paragraph. However, programming notation obscures the underlying

concepts, and English by itself is unsatisfactory. There is a simple language that is good for describing states and relations. It's the language used in just about every other science: mathematics. But what mathematics? Everyone who works on formalizing computation says that they use mathematics. Process algebra is algebra, which is certainly math. Category theory and temporal logic are also math. These esoteric forms of math have their place, but that place is not in the basic education of a computer engineer. The only mathematics we need to describe computations are sets, functions, and simple predicate logic.

For historical reasons, mathematicians use and teach mathematics in ways that are not well suited for computer science. For example, while it is crucial for computer engineers, an understanding of simple logic is not important for most mathematicians. Consequently, although it has been centuries since mathematicians wrote algrebraic formulas in words, they still usually obscure simple logical concepts like quantification by expressing them in prose. This is perhaps why many computer scientists feel that the standard method of formalizing ordinary mathematics used by logicians for almost a century, consisting of predicate (first-order) logic and elementary set theory, is inadequate for computer science. This is simply not true. Other, more complicated logical systems that introduce concepts such as types may be useful for some applications. However, there is no more need to use them in basic compute science education than there is to use them in teaching calculus (or arithmetic). Computer science should be based on the same standard mathematics as the rest of science.

Another problem with the way mathematicians use mathematics is its informality. Informal mathematics is fine for explaining concepts like states and relations, and for explaining invariants of simple algorithms like the one in the example above. However, a defining characteristic of computing is the need for rigor. Incorrectly writing $>$ instead of $\geq$ in the statement of a theorem is considered in mathematics to be a trivial mistake. In an algorithm, it could be a serious error. Almost no mathematicians know how to do mathematics with the degree of formal rigor needed to avoid such mistakes. They may study formal logic; they don't use it. Most think it impractical to do mathematics completely rigorously. I have often asked mathematicians and computer scientists the following question: How long would a purely formal definition of the Riemann integral (the definite integral of elementary calculus) be, assuming only the arithemtical operators on the real numbers and simple math? The answer usually ranges from 50 lines to 50 pages.

TLA$^+$ is one of those esoteric languages based on temporal logic [5]. However, if one ignores the TL (the temporal logic operators), one obtains

a language for ordinary math I will call here A-Plus. For example, here is an A-Plus definition of the operator GCD such that $GCD(m, n)$ equals the greatest common divisor of positive integers m and n.

$$
\begin{array}{lll}
GCD(m,n) & \triangleq & \text{LET } DivisorsOf(p) \triangleq \{d \in 1\,..\,p : \\
 & & \qquad \exists\, q \in 1\,..\,p : p = d * q\} \\
 & & MaxElementOf(S) \triangleq \text{CHOOSE } s \in S : \\
 & & \qquad \forall\, t \in S : s \geq t \\
 & & \text{IN } MaxElementOf(DivisorsOf(m) \cap DivisorsOf(n))
\end{array}
$$

A-Plus is obtained by deleting from TLA$^+$ all non-constant operators except the prime operator ($'$) that is used to express next-state relations—for example, $x' = 1 + x$ is a relation that is true iff the value of x in the next state equals 1 plus its value in the current state. A-Plus is a practical language for writing formal mathematics. An A-Plus definition of the Riemann integral takes about a dozen lines. One can use A-Plus to describe computations, and those descriptions can be executed by the TLC model checker to find and eliminate errors. Using A-Plus is a lot like programming, except it teaches users about math rather than about a particular set of programming language constructs.

A-Plus is not the best of all possible languages for mathematics. It has its idiosynchracies, and some people will prefer different ways of formalizing elementary mathematics—for example, Hilbert-Bernays rather than Zermelo-Fraenkel set theory. Any language for ordinary first-order logic and set theory will be fine, as long as it eschews complicated computer-science concepts like types and objects. However, it should have tools for checking descriptions of computations. Despite the inherent simplicity of mathematics, it's almost as hard to write error-free mathematical descriptions of computations as it is to write error-free programs.

Mathematics is an extremely powerful language, and it's a practical method for describing many classes of computing devices. However, programming language constructs such as assignment were developed for a reason. Although more complicated than ordinary mathematics, a simple language based on traditional programming constructs can be convenient for describing certain kinds of algorithms. Such a language is a useful tool rather than a barrier to understanding only if we realize that it is just a shorthand for writing mathematical descriptions of computations. The language must be simple enough that the user can translate an algorithm written in the language to the equivalent mathematical description. There are several rather simple languages that have this property—for example, the Unity program-

ming language. The most powerful such language I know of is PlusCal [4]. A PlusCal algorithm is essentially translated to an A-Plus description, which can be checked using the TLA$^+$ tools. Because any A-Plus expression can be used as an expression in a PlusCal algorithm, PlusCal's expression language has the full power of ordinary math, making PlusCal enormously expressive.

Students with an understanding of computations and how to express them mathematically and of invariance are ready to learn about concurrency. What they should be taught depends on their needs. If they need to learn how to write real concurrent programs, they will need to learn a real programming language. Programming languages are much more complicated than mathematics, and it is impractical to try to explain them precisely and completely. However, their basic features can be understood with the aid of mathematics. The difference between = and *.equals* in an object-oriented programming language is easily explained in terms of sets and functions.

Teaching concurrency generally includes teaching about concurrent algorithms. Distributed algorithms are usually easy to describe directly in mathematics, using a language like A-Plus. Multithreaded algorithms are usually more convenient to describe with the aid of programming constructs such as the ones in PlusCal. In any case, an algorithm should be explained in terms of its invariant. When you use invariance, you are teaching not just an algorithm, but how to think about algorithms.

Even more important than an algorithm is the precise specification of what the algorithm is supposed to do. A specification is a set of computations—namely, the set of all computations that satisfy the specification. An engineer can find algorithms in a textbook. A textbook won't explain how to figure out what problem a system needs to solve. That requires practice. Carefully specifying a problem before presenting an algorithm that solves it can teach an engineer how to understand and describe what a system is supposed to do.

Education is not the accumulation of facts. It matters little what a student knows after taking a course. What matters is what the student is able to do after taking the course. I've seldom met engineers who were hampered by not knowing facts about concurrency. I've met quite a few who lacked the basic skills they needed to think clearly about what they were doing.

References

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

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

[3] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[4] Leslie Lamport. The pluscal algorithm language. URL `http://research.microsoft.com/users/lamport/tla/pluscal.html`. The page can also be found by searching the Web for the 25-letter string obtained by removing the "`-`" from `uid-lamportpluscalhomepage`.

[5] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2003.