Distributed Computing *and* Education Column *Special Issue*

by

Juraj Hromkovic, Stefan Schmid

ETH Zurich, University of Vienna

In this column, Leslie Lamport makes the case for using the language of mathematics for describing algorithms. He argues that students should learn to think mathematically when writing code and programs.

The column is a special issue and collaboration of the distributed computing column and the education column.

Enjoy!

If You're Not Writing a Program, Don't Use a Programming Language

Leslie Lamport
Microsoft Research, Mountain View, CA, U.S.A.

Abstract

The need to handle large programs and to produce efficient compiled code adds complexity to programming languages and limits their expressiveness. Algorithms are not programs, and they can be expressed in a simpler and more expressive language. That language is the one used by almost every branch of science and engineering to precisely describe and reason about the objects they study: the language of mathematics. Math is useful for describing a more general class of algorithms than are studied in algorithm courses.

1 Introduction

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. I have found that most of them do not understand some important basic concepts. These concepts are obscured by programming languages. They are better understood by a simple and powerful way of thinking about computation mathematically that is explained here.

This note discusses what is called *correctness* in the field of program verification: the requirement that each possible execution of a program satisfies some precisely defined property. For brevity, I will use "correctness" only in this sense, ignoring other common meanings of the word. For example, ease of use is not a correctness condition because it isn't precisely defined. That a program produces the right output 99.9% of the time is also not a correctness condition because it isn't an assertion about an individual execution. However, producing the right output *is* a correctness condition. A mathematical understanding of this concept of correctness is useful beyond the field of program verification. It provides a way of thinking that can improve all aspects of writing programs and building systems.

I will consider algorithms, not programs. It's fruitless to try to precisely distinguish between them, but we all have a general idea that an algorithm is a higher-level abstraction that is implemented by a program. For example, here is Euclid's algorithm for computing $GCD(M, N)$, the greatest common divisor of positive integers M and N:

> Let x equal M and y equal N. Repeatedly subtract the smaller of x and y from the larger. Stop when x and y have the same value, at which point that value is $GCD(M, N)$.

Implementing this algorithm in a programming language requires adding details that are not part of the algorithm. For example, should the types of M and N be single-precision integers, double-precision integers, or some class of objects that can represent larger integers? Should the program assume that M and N are positive or should it return an error if they aren't?

The algorithms found in textbooks and studied in algorithm courses are relatively simple ones that are useful in many situations. Most non-trivial programs implement one or more algorithms that are used only in that program. Coding is the task of implementing an algorithm in a programming language. However, programming is too often taken to mean coding, and the algorithm is almost always developed along with the code. The programmer is usually unaware of the existence of the algorithm she is developing. To understand why this is bad, imagine trying to discover Euclid's algorithm by thinking in terms of code rather than in terms of mathematics.

The term *algorithm* is generally taken to mean only an algorithm that might appear in a textbook on algorithms. The special-purpose algorithms implemented by programs and systems are usually called something else, such as high-level designs, specifications, or models. However, they differ from textbook algorithms only in being special purpose and often more complicated. I will call them algorithms to emphasize that they are fundamentally the same as what are conventionally called algorithms.

The benefit of thinking about algorithms mathematically is not limited to obviously mathematical problems like computing the GCD. Here's what I was told in an email from the leader of a team that built a real-time operating system by starting with a high-level design written in a mathematics-based language called TLA$^+$ [14]:

> The [TLA$^+$] abstraction helped a lot in coming to a much cleaner architecture (we witnessed first-hand the brainwashing done by years of C programming). One of the results was that the code size is about 10$\times$ less than in [the previous version].

The previous version of the operating system was flown in a spacecraft, where one assumes reducing code size was important. The common obsession with languages might lead readers to think this result was due to some magical features of TLA$^+$. It wasn't. It was due to TLA$^+$ letting users think mathematically.

This note is written for sophisticated readers, but the mathematical approach it presents is simple enough to be understood by undergraduates. The mathematics can be made as rigorous or as informal as we want. The discussion here is informal, using only simple examples. Not explained are how to make the mathematics completely formal and how to handle the large formulas that describe complex real-world algorithms. Also not explained is how to write and reason about those mathematical formulas in practice. That would require a book.

2 Behaviors and Properties

An execution of an algorithm is represented mathematically as a sequence of states, where a state is an assignment of values to variables. A sequence of states is called a *behavior*. For example, the following three-state behavior represents the one possible execution of Euclid's algorithm for $M = 12$ and $N = 18$

$$[x \leftarrow 12,\ y \leftarrow 18]\ ,\ \ [x \leftarrow 12,\ y \leftarrow 6]\ ,\ \ [x \leftarrow 6,\ y \leftarrow 6]$$

where $[x \leftarrow 12,\ y \leftarrow 18]$ is the state that assigns 12 to x and 18 to y. The simplicity and power of this way of representing executions has led me to find it the best one for studying correctness (as defined above).

A *property* is a predicate (Boolean-valued function) on behaviors. We say that a behavior b *satisfies* a property P, or that P *is true on* b, iff (if and only if) $P(b)$ equals TRUE. A correctness condition of an algorithm asserts that every behavior that represents an execution of the algorithm satisfies a property.

Partial correctness of Euclid's algorithm means that if it stops, then x and y both equal $GCD(M, N)$. The algorithm stops iff x and y have the same value. Therefore, partial correctness of the algorithm is expressed by the property that is true of a behavior iff every state of the behavior satisfies this condition:

If x and y have the same value, then that value equals $GCD(M, N)$.

The state predicate (Boolean-valued function on states) that is true on a state iff the state satisfies this condition can be written as the formula

$$(x = y)\ \Rightarrow\ (x = GCD(M, N)) \tag{1}$$

where $\Rightarrow$ denotes logical implication. The property that is true on a behavior iff (1) is true on every state of the behavior is written as

$$\Box((x = y) \Rightarrow (x = GCD(M, N))) \tag{2}$$

For $M = 12$ and $N = 18$, property (2) is true on this five-state behavior:[1]

$$[x \leftarrow 1,\ y \leftarrow 37]\ ,\ \ [x \leftarrow 6,\ y \leftarrow 6]\ ,\ \ [x \leftarrow 42,\ y \leftarrow 7]\ ,$$
$$[x \leftarrow 6,\ y \leftarrow 6]\ ,\ \ [x \leftarrow 0,\ y \leftarrow 12]$$

Partial correctness of Euclid's algorithm means that property (2) is true of every behavior representing an execution of Euclid's algorithm. Amir Pnueli introduced $\Box$ into computer science as an operator of temporal logic [13], but we can consider it here to be an ordinary mathematical operator that maps state predicates to properties.

Observe that, like almost all mathematicians, I say that formula (2) *is* a property, which is a Boolean-valued function. Pedants and logicians might say that the property is the meaning of the formula, which is different from the formula itself. Like almost all mathematicians, I will ignore this distinction.

A property of the form $\Box I$ for a state predicate I is called an *invariance property*, and we say that $\Box I$ asserts that I is an *invariant*. Invariants play a crucial role in understanding algorithms.

Another important property of Euclid's algorithm is that it always terminates. The algorithm terminates when x equals y. Therefore, an execution that terminates is represented by a behavior containing a state in which the state predicate $x = y$ is true. A behavior contains such a state iff it is not the case that all of its states satisfy $x \neq y$—that is, iff the property $\Box(x \neq y)$ is not true on the behavior, which means that $\neg\Box(x \neq y)$ *is* true on the behavior. Hence a terminating behavior of the algorithm is one satisfying the property $\neg\Box(x \neq y)$.

3 Algorithms are Properties

We usually think of an algorithm as generating possible executions. For example, we can think of Euclid's algorithm generating an execution for each pair of positive integers M and N. (Nondeterminism is a more interesting source of multiple possible executions for an algorithm.) Each possible execution is represented mathematically by a behavior. We can represent an algorithm as the set of all these behaviors.

There is a natural correspondence between sets and predicates. A set S corresponds to the predicate, let's call it π_S, that is defined by letting $\pi_S(e)$ equal TRUE iff e is an element of S. Thus, instead of representing an algorithm by a set S of behaviors, we can represent it by the corresponding predicate π_S on behaviors. A predicate on behaviors is what we call a property, so corresponding to the set S of

[1]Note that a behavior is *any* sequence of states, not just one representing the execution of some algorithm.

behaviors representing executions of the algorithm is the property π_S that is true of a behavior iff the behavior represents an execution of the algorithm.

I will usually represent an algorithm with set S of behaviors as the property π_S because I find that to be the most helpful way of thinking about algorithms. This means that instead of thinking of an algorithm as a generator of executions, we think of it as a rule for determining if a sequence of states is an execution of the algorithm. (In computer science jargon, we think of the algorithm as a recognizer rather than a generator of executions.) This way of thinking about algorithms may strike you as either bizarre or an insignificant shift of viewpoint. However, I have found it to be quite helpful for understanding algorithms.

Algorithms are properties, but not every property is an algorithm. We use the term algorithm for a property that is satisfied by a set of behaviors representing the possible executions of what we think of as an algorithm.

We saw in Section 2 that a correctness condition of an algorithm asserts that every behavior representing an execution of the algorithm satisfies a property P. Since the algorithm is a property A, this correctness condition asserts that if a behavior satisfies A then it satisfies P — an assertion written mathematically as $A \Rightarrow P$.[2] The property P could be an algorithm that is a higher-level (more abstract) version of the algorithm A. In that case, we say that $A \Rightarrow P$ means that A *refines* P. Thus, our mathematical view of computation provides a natural definition of algorithm refinement as implication. Refinement is generalized in Section 6.1 to include data refinement [6].

4 Describing Algorithms Mathematically

4.1 State Machines

The practical way to precisely describe algorithms is with state machines. A state machine is usually described by a set of possible initial states and a next-state relation that determines the possible steps, where a step is a pair of successive states in a behavior. The possible executions of the state machine consist of all sequences $s_1, s_2, \ldots$ of states such that (1) s_1 is a possible initial state and (2) every step (s_i, s_{i+1}) satisfies the next-state relation. A Turing machine is obviously a state machine. An operational semantics of a programming language describes every program in the language as a state machine. Nondeterminism is represented by a next-state relation that is satisfied by more than one pair (s, t) of states for

[2]I am extending the operator $\Rightarrow$ on Boolean values to an operator on Boolean-valued functions. Such extensions are ubiquitous in mathematics. For example, if f and g are numerical-valued functions, then $f + g$ is the function defined by $(f + g)(x) = f(x) + g(x)$.

the same state s. We'll see in Section 4.4 that this definition of a state machine is incomplete, but it will suffice for now.

If a state is an assignment of values to variables, then an execution of a state machine is a behavior. The set of initial states can be represented as a state predicate and the next-state relation can be represented as a predicate on pairs of states. The state machine is represented as a property that is true on a behavior $s_1, s_2, \ldots$ iff these two conditions are satisfied:

SM1 The initial-state predicate is true on s_1.

SM2 The next-state predicate is true on every step (s_i, s_{i+1}).

4.2 State Machines in Mathematics

Let's represent Euclid's algorithm by a state machine. We described the algorithm above for a single pair of input values M and N, so we considered Euclid's algorithm for different values of M and N to be different algorithms. Let's do this in our mathematical representation of the algorithm, so we describe Euclid's algorithm for a single fixed pair M, N of positive integers. The algorithm has a single behavior; what that single behavior is depends on the (unspecified) values of the positive integers M and N.

The initial predicate is true on a state iff x has the value M and y has the value N. This state predicate can obviously be written

$$(x = M) \land (y = N) \tag{3}$$

A predicate on pairs of states is often written as a formula containing primed and unprimed variables, where an unprimed variable represents the value of the variable in the first state and a primed variable represents its value in the second state [5]. With this notation, the next-state predicate for Euclid's algorithm is

$$\begin{array}{ll} & (\quad (x > y) \\ & \quad \land\ (x' = x - y) \\ & \quad \land\ (y' = y) \quad) \\ \lor & (\quad (y > x) \\ & \quad \land\ (y' = y - x) \\ & \quad \land\ (x' = x) \quad) \end{array} \tag{4}$$

For example, the predicate (4) is true on the pair of states

$$(\,[x \leftarrow 18,\ y \leftarrow 12],\ [x \leftarrow 6,\ y \leftarrow 12]\,)$$

because that formula equals TRUE if $x = 18$, $y = 12$, $x' = 6$, and $y' = 12$. Formula (4) equals FALSE if the value of x equals the value of y. Hence, the next-state predicate of Euclid's algorithm equals FALSE for any pair of states with $x = y$ true in the first state. This means that Euclid's algorithm halts in such a state.

Let's define $Init_E$ to equal the initial-state predicate (3) and $Next_E$ to equal the next-state predicate (4). These two formulas describe Euclid's algorithm—that is, they describe the property that is our mathematical representation of Euclid's algorithm. It's more convenient to describe this property with a single formula, which we now do.

First, define $\flat P$, for any state predicate P, to be the property that is true on a behavior iff P is true on the first state of the behavior. Then $\flat Init_E$ is the property that expresses condition SM1 of the state machine.

Next, extend the operator $\Box$ to predicates Q on pairs of states by letting $\Box Q$ be true on a behavior iff Q is true on every step of the behavior. Thus, $\Box Next_E$ is the property that expresses condition SM2 of the state machine.

The property that is Euclid's algorithm, which is the property satisfying conditions SM1 and SM2, is represented by the formula $\flat Init_E \land \Box Next_E$. For a state predicate P, it's customary to write $\flat P$ as simply P, determining from context whether P means the state predicate or the property $\flat P$. We thus write the property that is Euclid's algorithm as

$$Init_E \ \land\ \Box Next_E \tag{5}$$

Instead of defining Euclid's algorithm for a specific pair of integers, we could define it for an arbitrary pair of integers. The definition is the formula obtained by existentially quantifying formula (5) over all positive integers M and N — a formula I will write

$$\exists\ M, N \in \mathbf{Z}^+ \bullet (Init_E \ \land\ \Box Next_E)$$

where $\mathbf{Z}^+$ is the set of positive integers. Since M and N don't occur in $Next_E$, this formula can also be written

$$(\exists\ M, N \in \mathbf{Z}^+ \bullet Init_E) \ \land\ \Box Next_E$$

However, let's stick with our definition (5) of Euclid's algorithm for a single pair of positive integers M and N.

4.3 Proving Invariance

Let's now prove partial correctness of Euclid's algorithm, which is expressed as the property (2). We first consider the general problem of proving that an algorithm described by the formula $Init \land \Box Next$ satisfies an invariance property $\Box I$. We saw in Section 3 that this means proving the formula $Init \land \Box Next \Rightarrow \Box I$.

Since $\Box I$ asserts that the state predicate I is true on all states of a behavior, the natural way to prove $\Box I$ is by induction: proving that (i) I is true on the first state and (ii) for any j, if I is true on the j^{th} state then it's true on the $(j+1)^{\text{st}}$ state.

We can obviously prove (i) by proving $Init \Rightarrow I$. To prove (ii), it suffices to prove that for any pair (s, t) of states, if I is true on s and $Next$ is true on (s, t), then I is true on t. Let's define I' to be the formula obtained from I by priming all its variables. The formula I' represents the predicate on pairs of states that is true on (s, t) iff I is true on t. We can therefore prove (ii) by proving that I true on s and N true on (s, t) implies that I' is true on (s, t) — an assertion expressed by the formula $I \wedge Next \Rightarrow I'$. We can encapsulate all this in the following proof rule, which asserts that the truth of the two formulas above the line (the hypotheses) implies the truth of the formula below the line (the conclusion).

$$\frac{\begin{array}{l} Init \Rightarrow I \\ I \wedge Next \Rightarrow I' \end{array}}{Init \wedge \Box Next \Rightarrow \Box I} \tag{6}$$

A formula I satisfying the hypotheses of this rule is called an *inductive invariant* of the algorithm $Init \wedge \Box Next$. An inductive invariant is an invariant, but the converse isn't necessarily true. The invariant (1) of Euclid's algorithm is not an inductive invariant of the algorithm—for example, if $M = 12$ and $N = 18$, then the second hypothesis equals FALSE when $x = 16$ and $y = x' = y' = 8$. Define Inv_E to equal formula (1). To prove that Inv_E is an invariant, we find an inductive invariant I that implies it. The invariance of Inv_E then follows from the invariance of I and the following proof rule, which asserts the obvious fact that if a state predicate P implies a state predicate Q, then P true on all states of a behavior implies that Q is true on all states of the behavior.

$$\frac{P \Rightarrow Q}{\Box P \Rightarrow \Box Q} \tag{7}$$

The inductive invariant I for Euclid's algorithm is

$$GCD(x, y) = GCD(M, N)$$

The first hypothesis of rule (6) is trivially true. The truth of the second hypothesis follows from the observation that for any integers a and b, an integer divides both a and b iff it divides both a and $a - b$. That I implies Inv_E is obvious because $GCD(x, x)$ equals x.

As illustrated by Euclid's algorithm, partial correctness is an invariance property. The Floyd/Hoare method of proving partial correctness is based on proof rules (6) and (7), where the inductive invariant is written as a program annotation [3, 7].

What an algorithm does next is determined by its current state, not by what happened in the past. An algorithm does the right thing (for example, it terminates only with the right answer) because something is true of every state—that is, because it satisfies an invariance property. Understanding an algorithm requires understanding that invariant. Years of experience has shown that the one reliable method of proving the invariance of a state predicate is by finding an inductive invariant that implies it.

4.4 Termination

For $M = 12$ and $N = 18$, the formula $Init_E \wedge \Box Next_E$ is satisfied by the following three behaviors:

1. $[x \leftarrow 12,\ y \leftarrow 18]$
2. $[x \leftarrow 12,\ y \leftarrow 18],\ [x \leftarrow 12,\ y \leftarrow 6]$
3. $[x \leftarrow 12,\ y \leftarrow 18],\ [x \leftarrow 12,\ y \leftarrow 6],\ [x \leftarrow 6,\ y \leftarrow 6]$

Those are the three behaviors that satisfy conditions SM1 and SM2. (Behavior 1 trivially satisfies SM2 because it has no steps.) We consider behavior 3 to be the only correct one; we don't want Euclid's algorithm to allow behaviors 1 and 2, which stop prematurely.

A common approach is to modify the definition of the executions of a state machine by adding another condition to SM1 and SM2. Define a predicate N on pairs of states to be *enabled* on a state s iff there is some state t such that N is true on (s, t). The additional condition is:

SM3 The behavior does not end in a state in which the next-state predicate is enabled.

The definition of $\Box Next$ could be modified to imply SM3. While this approach is satisfactory for sequential algorithms that compute an answer and then stop, we'll see in Section 5.1 that it's a bad idea for more general algorithms, including concurrent ones.

Instead, we express the requirement that Euclid's algorithm not stop before it should by adding the requirement of weak fairness of the next-state relation—expressed by the formula WF($Next_E$). For any predicate N on pairs of states, WF(N) is true on a finite behavior iff the behavior doesn't end in a state in which N is enabled. It is true on an infinite behavior iff the behavior doesn't end with an infinite sequence of states such that (i) N is enabled on all states of the sequence and (ii) N is not true on any of that sequence's steps.

Euclid's algorithm terminates iff it eventually reaches a state in which $x = y$ is true, and we saw in Section 2 that this is asserted by the property $\neg\Box(x \neq y)$.

Thus, the assertion that every behavior of Euclid's algorithm stops is expressed mathematically by:

$$Init_E \ \wedge\ \Box Next_E \ \wedge\ \mathrm{WF}(Next_E) \ \ \Rightarrow\ \ \neg\Box(x \neq y) \tag{8}$$

To prove (8) we prove two things:

1. $x + y$ is non-negative in all states of a behavior of the algorithm, which is expressed by:

$$Init_E \ \wedge\ \Box Next_E \ \ \Rightarrow\ \ \Box(x + y \geq 0) \tag{9}$$

2. Executing a step of the algorithm from a state with $x \neq y$ decreases $x + y$, which is implied by

$$(x \neq y) \ \wedge\ Next_E \ \ \Rightarrow\ \ (x' + y') < (x + y) \tag{10}$$

We've seen how to prove an invariance property like (9). Formula (10) follows from the fact that $(a - b) + b < a + b$ for any positive numbers a and b.[3] Since a non-negative integer can be decreased only a finite number of times before it becomes non-positive, and $Next_E$ is enabled iff $x \neq y$, (9) and (10) imply (8).

Termination of any sequential algorithm is proved in this way. In general, the formula $x + y$ is replaced by a suitable integer-valued function on states (usually called a *variance* function), and $x \neq y$ is replaced by the state predicate asserting that the next-state predicate is enabled.

5 Concurrent Algorithms

Many computer scientists seem to believe that a concurrent algorithm must be described by a collection of state machines that communicate in some way, each state machine representing a separate process. In fact, a concurrent algorithm is better understood as a single state machine. I will illustrate this with a simple example: the N-buffer producer/consumer algorithm.

5.1 The Two-Process Algorithm

In the standard description of the algorithm, a producer process sends a sequence of messages that are received by a consumer process. Messages are transmitted using an array of N message buffers numbered 0 through $N - 1$, with $N > 0$.

[3] A more rigorous proof requires showing that $(x > 0) \wedge (y > 0)$ is an invariant of Euclid's algorithm.

The producer sends messages by putting them in successive buffers starting with buffer 0, putting the i^{th} message in buffer $i-1$ mod N; and the consumer removes messages in the same order from the buffers. The producer can put a message in a buffer only if that buffer is empty, and the consumer can remove a message only from a buffer that contains one.

The algorithm is described by the formula

$$Init_{PC} \;\wedge\; \Box Next_{PC} \;\wedge\; F_{PC} \tag{11}$$

where $Init_{PC}$ is the initial-state predicate, $Next_{PC}$ is the next-state predicate, and F_{PC} is a fairness property that is false on a behavior that ends before it should. I won't write the complete definitions of these formulas and will just briefly discuss $Next_{PC}$ and F_{PC}.

The obvious definition of $Next_{PC}$ has the form

$$Next_{PC} \;\equiv\; Send \;\vee\; Rcv$$

where $Send$ is true on a pair (s, t) of states iff t represents the state obtained from s by having the producer put its next message into a buffer; and similarly Rcv is true on a pair of states iff that pair represents the consumer removing its next message from a buffer.

There are three obvious choices for the property F_{PC}, leading to three different algorithms. We might want to require that messages keep getting sent and received forever. This is expressed by letting F_{PC} equal WF($Next_{PC}$). We can write the same algorithm by letting F_{PC} equal WF($Send$) $\wedge$ WF(Rcv). These two formulas are not equivalent, but they produce equivalent formulas (11) because this formula is true:

$$Init_{PC} \;\wedge\; \Box Next_{PC} \;\Rightarrow\; (\mathrm{WF}(Next_{PC}) \;\equiv\; \mathrm{WF}(Send) \wedge \mathrm{WF}(Rcv))$$

Of course, we can't prove this formula without knowing the definitions of $Init_{PC}$, $Send$, and Rcv. However, it's possible to see from the definition of WF in Section 4.4 that it should be true because the informal description of the algorithm means $Init_{PC} \wedge \Box Next_{PC}$ should imply of a behavior that:

- $Send$ or Rcv is enabled in every state. (They can both be enabled.)
- The behavior can contain at most N consecutive steps that represent the sending of a message, and at most N consecutive steps that represent the receiving of a message.

The second obvious choice for F_{PC} requires that the producer should keep trying to send messages, but the consumer may stop receiving them. This is expressed by letting F_{PC} equal WF($Send$). Formula (11) then allows (but doesn't require) a behavior to end only in a state in which all message buffers are full. The third obvious choice is to let F_{PC} equal WF(Rcv), allowing behaviors to end only in a state in which all the buffers are empty. Note that modifying the definition of $\Box Next_{PC}$ by adding condition SM3 of Section 4.4 would make it impossible to write the last two versions of the algorithm.

5.2 Another View of the Algorithm

Because producer/consumer is a two-process algorithm, it's natural to write $Next_{PC}$ as the disjunction $Send \lor Rcv$ of two formulas, each describing the steps that can be taken by one of the processes. Let's take a closer look at those formulas. Sensible definitions of $Send$ and Rcv will have the form

$$\begin{aligned} Send &\equiv \exists\, i \in \{0, \ldots, N-1\} \bullet S(i) \\ Rcv &\equiv \exists\, i \in \{0, \ldots, N-1\} \bullet R(i) \end{aligned}$$

where $S(i)$ describes a step in which the producer puts a message in buffer i, and $R(i)$ describes a step in which the consumer removes a message from buffer i. Define $SR(i)$ to equal $S(i) \lor R(i)$. Elementary logic shows that $Next_{PC}$ equals

$$\exists\, i \in \{0, \ldots, N-1\} \bullet SR(i) \tag{12}$$

Formula (12) looks like the next-state predicate of an N-process algorithm, where the processes are numbered 0 through $N-1$ and $SR(i)$ describes steps that can be performed by process number i. Process i can put a message into buffer i when it's empty, and it removes a message from that buffer when it contains one.

Is the producer/consumer algorithm a 2-process algorithm or an N-process algorithm? Mathematically, it's neither. The algorithm is the formula/property (11). We can view that formula as a 2-process algorithm or an N-process algorithm. Each view gives us a different way of thinking about the algorithm, enabling us to understand it better than we could with just one view.[4]

Formula (4), the next-state predicate of Euclid's algorithm, is also the disjunction of two formulas. This means we can view Euclid's algorithm not just as a uniprocess algorithm, but also as a 2-process algorithm. One process tries to subtract y from x, which it can do only when $y < x$ is true; the other process tries to subtract x from y, which it can do only when $x < y$ is true.

[4]I've ignored the fairness property, so you may be tempted to think that F_{PC} will tell us how many processes there are in the algorithm. It won't. For example, the algorithm we get by letting F_{PC} equal WF($Send$) is also obtained by letting it equal

$$\forall\, i \in \{0, \ldots, N-1\} \bullet \mathrm{WF}(S(i))$$

This formula looks like the conjunction of fairness conditions on N separate processes.

All programming languages that I know of force you to think of the producer/consumer algorithm as either a 2-process algorithm or an N-process algorithm. Few algorithms are better understood by decomposing them into processes in different ways. But describing an algorithm with a programming language can limit our ability to understand it in other ways as well.

5.3 Safety and Liveness

What does correctness of the producer/consumer algorithm mean? Since the algorithm need not terminate, it doesn't satisfy any partial correctness condition or termination requirement. There's an endless variety of correctness properties that we might want to be satisfied by algorithms that need not terminate. Here are two that we might require of the version of the algorithm with F_{PC} equal to WF(Rcv).

PC1 The sequence of messages received by the consumer is a prefix of the sequence of messages sent by the producer.

PC2 Every message sent is eventually received.

PC1 is a *safety* property, which intuitively is a property asserting what *may* happen. PC2 is a *liveness* property, which intuitively is a property asserting what *must* happen. Here are the precise definitions. (Remember that every sequence of states is a behavior.)

- A safety property is one that is false on an infinite behavior iff it is false on some finite prefix of the behavior.
- A liveness property is one which, for any finite behavior, is true for some (possibly infinite) extension of that behavior.

It can be shown that any property is equivalent to the conjunction of a safety property and a liveness property. For any predicate I on states and predicate A on pairs of states, the formula $I \wedge \Box A$ is a safety property, and WF(A) is a liveness property.

The producer/consumer algorithm satisfies property PC1 iff $Init_{PC} \wedge \Box Next_{PC}$ implies that property. The fairness condition F_{PC} is irrelevant for safety properties. In general, for any algorithm $Init \wedge \Box Next \wedge F$, if F is the conjunction of formulas WF(A) and each of the predicates A implies $Next$, then the algorithm satisfies a safety property iff $Init \wedge \Box Next$ satisfies the safety property.[5]

[5]This is why we conjoin fairness conditions rather than other kinds of liveness conditions to $Init \wedge \Box Next$. If F were not of this form, the formula $Init \wedge \Box Next \wedge F$ would be hard to understand because the liveness property F could forbid steps allowed by $Next$.

5.3.1 Proving Safety Properties

An invariance property $\Box I$ is a safety property, and we've seen how to prove them. Stating PC1 as an invariance property would require being able to express the sequences of all messages that have been sent and received in terms of the formula's variables. Whether this is possible depends on the definitions of $Init_{PC}$, $Send$, and Rcv. If it's not possible, there are two ways to proceed.

The simplest approach is to add a history variable to the producer/consumer algorithm that records the sequences of messages sent and received, so PC1 can be expressed as an invariance property. A history variable is a simple kind of auxiliary variable—a variable that is added to an algorithm to produce a new algorithm that is the same as the original one if the value of the added variable is ignored [1]. (Auxiliary variables have other uses that I won't discuss.) In general, any safety property can be expressed as an invariance property by adding a history variable. However, this is often not a good approach because it encodes in an invariance proof the kind of unreliable behavioral reasoning that invariance proofs were developed to replace.

The second approach is to write a higher-level algorithm that obviously implies PC1, and then prove that the producer/consumer algorithm refines that algorithm. The higher-level algorithm will of course have the form $Init \land \Box Next$; the proof will use an invariance property $\Box Inv_{PC}$ of the producer/consumer algorithm and the following proof rule.

$$\frac{\begin{array}{l} Init_1 \;\Rightarrow\; Init_2 \\ Next_1 \land Inv_1 \land Inv_1' \;\Rightarrow\; Next_2 \end{array}}{Init_1 \land \Box Next_1 \land \Box Inv_1 \;\Rightarrow\; Init_2 \land \Box Next_2}$$

5.3.2 Proving Liveness Properties

Proving liveness properties like PC2 requires more than the simple counting-down argument used to prove termination. There is no single recipe for proving all liveness properties. Rigorous proofs are best done by generalizing $\Box$ to an operator on properties, where $\Box P$ asserts of a behavior that property P is true on all suffixes of that behavior. For example, $\Box\neg\Box(x \neq y)$ is true on an infinite behavior iff $x = y$ is true on infinitely many of its states. Weak fairness can be expressed with $\Box$, as can another important type of liveness property that I won't discuss called strong fairness [4].

At the heart of most liveness proofs are counting down arguments. The counting down argument used to prove termination of Euclid's algorithm is based on the fact that there is no infinite descending sequence $n_1 > n_2 > \ldots$ of natural numbers. Proving liveness properties of concurrent algorithms requires a generalization to counting down on a well-ordered set, which is a set S with partial order $\succ$ containing no infinite descending sequence $s_1 \succ s_2 \succ \ldots$ of elements. For example, the set of all k-tuples of natural numbers, with the lexicographical ordering, is well ordered.

We can regard $\Box$ as an ordinary mathematical operator on properties. For completely formal reasoning, I find it better to use temporal logic and to regard $\Box$ as a temporal operator. The proofs of liveness one writes in practice can be formalized with a small number of temporal-logic axioms [9]. However, temporal logic is a modal logic and does not obey some important laws of traditional math, so it must be used with care. An informal approach that avoids temporal logic may be best for undergraduates.

6 Refinement

6.1 Data Refinement

Data refinement is a traditional method of refining sequential algorithms that compute an output as a function of their input [6]. An example of data refinement is refining Euclid's algorithm by representing the natural numbers x and y with bit arrays ax and ay of length k.

Mathematically, a k-bit array a is a function from $\{0, \ldots, k-1\}$ to $\{0, 1\}$. It represents the natural number $AtoN(a)$, defined by

$$AtoN(a) \;\triangleq\; \sum_{i=0}^{k-1} a(i) \cdot 2^i$$

Let's write such a bit array a as $a(k-1) \ldots a(0)$, so $AtoN(01100)$ equals 12 for $k = 5$.

Let Alg_E be the formula $Init_E \wedge \Box Next_E \wedge WF(Next_E)$ that is Euclid's algorithm, and let Alg_{AE} be an algorithm whose variables are 5-bit arrays ax and ay. For any state s that assigns values to ax and ay, let $AEtoE(s)$ be the state that assigns the value $AtoN(ax)$ to x and $AtoN(ay)$ to y. For example,

$$AEtoE(\,[ax \leftarrow 01100,\ ay \leftarrow 10010]\,) \;=\; [x \leftarrow 12,\ y \leftarrow 18]$$

We extend $AEtoE$ to behaviors by defining

$$AEtoE(s_1, s_2, \ldots) \;\triangleq\; AEtoE(s_1),\ AEtoE(s_2),\ \ldots$$

For example, if b is the behavior

$$\begin{array}{l}[ax \leftarrow 01100,\ ay \leftarrow 10010],\ [ax \leftarrow 01100,\ ay \leftarrow 00110], \\ \quad [ax \leftarrow 00110,\ ay \leftarrow 00110]\end{array} \tag{13}$$

then $AEtoE(b)$ equals

$$[x \leftarrow 12,\ y \leftarrow 18],\ [x \leftarrow 12,\ y \leftarrow 6],\ [x \leftarrow 6,\ y \leftarrow 6] \qquad (14)$$

We say that algorithm Alg_{AE} *refines* algorithm Alg_E under the *refinement mapping* $x \leftarrow AtoN(ax)$, $y \leftarrow AtoN(ay)$ iff, for every behavior b allowed by algorithm Alg_{AE}, the behavior $AEtoE(b)$ is allowed by algorithm Alg_E. For $M = 12$ and $N = 18$, algorithm Alg_E allows only the single behavior (14), so if Alg_{AE} refines Alg_E, it will allow only the single behavior (13).

The value of the state predicate $x < 2 \cdot y$ on a state $AEtoE(s)$ equals the value of the predicate $AtoN(ax) < 2 \cdot AtoN(bx)$ on the state s. For example, both the value of $x < 2 \cdot y$ on the state

$$[x \leftarrow AtoN(10010),\ y \leftarrow AtoN(01100)]$$

and the value of $AtoN(ax) < 2 \cdot AtoN(bx)$ on the state

$$[ax \leftarrow 10010,\ ay \leftarrow 01100]$$

equal $18 < 2 \cdot 12$ (which equals TRUE). In general, the value of any state predicate I on $AEtoE(s)$ is the same as the value on s of the formula obtained by substituting $AtoN(ax)$ for x and $AtoN(ay)$ for y in I. Mathematicians have no standard notation for such a formula obtained by substitutions; I'll write it

$$I \text{ WITH } x \leftarrow AtoN(ax),\ y \leftarrow AtoN(ay) \qquad (15)$$

So, the value of I on $AEtoE(s)$ equals the value of formula (15) on s. Similarly, the value of the algorithm/formula Alg_E on a behavior $AEtoE(b)$ is the same as the value of

$$Alg_E \text{ WITH } x \leftarrow AtoN(ax),\ y \leftarrow AtoN(ay) \qquad (16)$$

on the behavior b. We said above that Alg_{AE} refines Alg_E under the refinement mapping described by this WITH clause iff, for any behavior b satisfying Alg_{AE}, the behavior $AEtoE(b)$ satisfies Alg_E. We've just seen that the latter condition is equivalent to b satisfying (16). Therefore, Alg_{AE} refines Alg_E under this refinement mapping iff Alg_{AE} implies (16). In general, we have:

> **Definition** Algorithm Alg_1 implements algorithm Alg_2 under the refinement mapping $v_1 \leftarrow e_1, \ldots\ v_n \leftarrow e_n$ iff this formula is true:
>
> $$Alg_1 \ \Rightarrow\ (Alg_2 \text{ WITH } v_1 \leftarrow e_1, \ldots, v_n \leftarrow e_n)$$

"The brainwashing done by years of C programming" may lead one to think that there is little difference between the expression $x' = x - y$ in the next-state relation (4) of Euclid's algorithm and the C assignment statement `x = x-y`. However, expanding the definition of Alg_E shows that formula (16) is an algorithm whose next-state predicate contains the expression

$$(x' = x - y) \text{ WITH } x \leftarrow AtoN(ax),\ y \leftarrow AtoN(ay)$$

which equals

$$\sum_{i=0}^{k-1} ax'(i) \cdot 2^i = \sum_{i=0}^{k-1} ax(i) \cdot 2^i - \sum_{i=0}^{k-1} ay(i) \cdot 2^i$$

No programming language allows you to write anything resembling this formula.

Data refinement is described by substitution, which is a fundamental operation of mathematics. It cannot be properly understood in terms of the limited kinds of substitution provided by programming languages.

6.2 Step Refinement

Another kind of refinement is step refinement, in which a single step of a high-level algorithm is refined by multiple steps of a lower-level algorithm. Let's consider a very simple example.

Suppose we want to write a formula/algorithm/property representing a clock that displays the hour and minute, ignoring the relation between the display and physical time. We could write a formula Alg_{HM} containing the variables hr and min that represent the hour and minute displays. A behavior satisfying Alg_{HM} would contain this subsequence of three states:

$$[hr \leftarrow 4,\ min \leftarrow 58],\ \ [hr \leftarrow 4,\ min \leftarrow 59],\ \ [hr \leftarrow 5,\ min \leftarrow 0]$$

Suppose we also write a formula Alg_{HMS} describing a clock that represents the hour, minute, and second displays with variables hr, min, and sec.

If we ask for a clock that displays hours and minutes, without explicitly saying that it does not display seconds, then our request is satisfied by a clock displaying hours, minutes, and seconds. In mathematics, writing a formula like Alg_{HM} containing the variables hr and min doesn't imply that there is no variable sec. The formula simply says nothing about sec or any other variable besides hr and min. Therefore, the formula/property Alg_{HM} should be satisfied by the algorithm/property Alg_{HMS}. In other words, every behavior satisfying Alg_{HMS} should also satisfy Alg_{HM}. However, the way I've been writing our algorithms, a behavior satisfying Alg_{HMS} takes 60 steps to go from a state with $hr = 4$ and $min = 59$

to one with $hr = 5$ and $min = 0$, while a behavior satisfying Alg_{HM} does it in a single step. Therefore, I haven't been writing algorithms the way they should be written. Writing them properly requires a closer look at how mathematics is used to describe the world.

I defined a state to be an assignment of values to variables, and in the examples I've taken the variables to be the ones in the algorithm. Since writing the formula Alg_{HM} doesn't preclude the existence of variables other than hr and min, for what we are doing to make sense mathematically, a state should be an assignment of values to all possible variables. (Mathematicians assume that there are an infinite number of possible variables.) The formula Alg_{HM} is not an assertion about a universe consisting only of an hour-minute clock described by the variables hr and min. It's an assertion about a universe containing an hour-minute clock—a universe that might also contain Euclid's algorithm and the producer/consumer algorithm. A behavior represents a possible "execution" of this entire universe. The behavior satisfies formula Alg_{HM} iff it represents a universe in which the hour-minute clock is operating correctly. If Alg_{PC} is a specification of the producer/consumer algorithm, then a behavior satisfies $Alg_{HM} \land Alg_{PC}$ iff it represents a universe in which both the hour-minute clock and the producer/consumer algorithm are operating correctly.

It's obviously absurd for a specification of the hour-minute clock to require that, in a state with $hr = 4$ and $min = 59$, the next state of the entire universe must be one with $hr = 5$ and $min = 0$. It should allow multiple successive states with $hr = 4$ and $min = 59$ to precede a state with $hr = 5$ and $min = 0$ — perhaps trillions of them. This means that the next-state predicate for the hour-minute clock should have the form

$$Tick_{HM} \lor (hr' = hr \land min' = min) \tag{17}$$

where $Tick_{HM}$ describes how hr and min can change. Steps that leave hr and min unchanged (allowed by the second disjunct) are called *stuttering steps* of the algorithm. Steps allowed by Alg_{HMS} that change only sec are stuttering steps of Alg_{HM}, allowed by the next-state predicate (17). Therefore, Alg_{HMS} will imply Alg_{HM}.

All the formulas representing algorithms that we've written need to be modified to allow stuttering steps. Let's write formula (17) as $[Tick]_{\langle hr,min\rangle}$. We can then change the safety part (5) of Euclid's algorithm to $Init_E \land \Box[Next_E]_{\langle x,y\rangle}$ and the safety part of the producer/consumer algorithm (11) to $Init_{PC} \land \Box[Next_{PC}]_{\langle\ldots\rangle}$, where "..." is the list of all the algorithm's variables.

The next-state predicate (17) allows behaviors that, from some point on, contain only stuttering steps of the clock. Such a behavior represents one in which the clock stops. Since the entire universe need never stop, termination of any algorithm is represented by infinite stuttering. We can therefore simplify the mathematics by considering only infinite behaviors. Termination is still disallowed by fairness properties. The fairness condition WF($Tick_{HM}$) asserts that the hour-minute clock never stops, assuming that $Tick_{HM}$ does not allow steps that leave both hr and min unchanged.

In general, stuttering steps allow step refinement in which one step of a higher-level version of an algorithm is implemented by multiple steps of a lower-level version. One of those lower-level steps allows the higher-level step; the rest allow stuttering steps of the higher-level algorithm.

6.3 Proving Correctness by Refinement

As we have seen in Section 5.3.1, a correctness property of an algorithm is often best expressed as a higher-level algorithm. Proving correctness then means proving that the original algorithm refines the higher-level one. This usually involves both data refinement and step refinement. For example, an algorithm that refines Euclid's algorithm by representing integers with bit strings might refine a step of Euclid's algorithm with a sequence of steps that read or modify only a single bit at a time. The refinement mapping must be defined so that only one of those steps refines a step of Euclid's algorithm that modifies x or y. The rest must refine stuttering steps.

I expect that this kind of refinement sounds like magic to most readers, who won't believe that it can work in practice. Seeing that it is a straightforward, natural way to reason about algorithms requires working out examples, which I will not attempt here. I will simply report that among the refinement proofs I have written is a machine-checked correctness proof [11] of the consensus algorithm at the heart of a subtle fault-tolerant distributed algorithm by Castro and Liskov [2] that uses $3F + 1$ processes, up to F of which may be malicious (Byzantine). The proof shows that the Castro-Liskov consensus algorithm refines a version of the $2F + 1$ process Paxos consensus algorithm that tolerates F benignly faulty processes [10]. Steps of malicious processes, as well as many steps taken by the good processes to prevent malicious ones from causing an incorrect execution of Paxos, refine stuttering steps of the Paxos algorithm. I found that viewing the Castro-Liskov algorithm as a refinement of Paxos was the best way to understand it.

7 Conclusion

Algorithms are usually described with programming languages or languages based on programming-language concepts. The mathematical approach presented here can be viewed as describing algorithms semantically. It may seem impractical to people used to thinking in terms of programming languages, whose semantics are so complicated. But programming languages are complicated because programs can be very large and must be executed efficiently. Algorithms are much smaller than programs, and they don't have to be executed efficiently.[6] This makes it practical to describe them in the much simpler and infinitely more expressive language of mathematics.

The informal mathematics I have used has not been rigorous. For example, $GCD(x, y) = GCD(M, N)$ is not really an inductive invariant of Euclid's algorithm. To make it inductive, we must conjoin the assertion that x and y are integers. A completely rigorous exposition might be inappropriate for undergraduates. However, their professors should understand how to reason rigorously about algorithms.

A simple formal basis for mathematics, developed about a century ago and commonly accepted by mathematicians today, is first-order logic and (untyped) set theory. To my knowledge, this is an adequate formalization of the mathematics used by scientists and engineers. (It has been found inadequate for formalizing the long, complicated proofs mathematicians can write.) Many computer scientists feel that types are necessary for rigor. Besides adding unnecessary complexity, types can introduce problems for mathematical reasoning that become evident only when one tries to provide a formal semantics for the language being used—something textbook writers rarely do.

The ideas put forth here are embodied in the TLA$^+$ specification language [8] mentioned in the introduction. TLA$^+$ is a formal language with tools that include a model checker and a proof checker. It was designed for describing concurrent algorithms, including high-level designs of distributed systems. Any attempt to formalize mathematics in a practical language requires choices of notation and underlying formalism that will not please everyone. Moreover, languages and tools that are better than TLA$^+$ for other application domains should be possible. But TLA$^+$ demonstrates that the approach described here is useful in engineering practice [12].

Today, programming is generally equated with coding. It's hard to convince students who want to write code that they should learn to think mathematically, above the code level, about what they're doing. Perhaps the following observation will give them pause. It's quite likely that during their lifetime, machine learning will completely change the nature of programming. The programming languages they are now using will seem as quaint as Cobol, and the coding skills they are learning will be of little use. But mathematics will remain the queen of science, and the ability to think mathematically will always be useful.

[6]Tools for checking an algorithm may have to execute it, but the execution need not be as efficient as that of a program implementing the algorithm.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186. ACM, 1999.

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

[4] Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, New York, Berlin, Heidelberg, Tokyo, 1986.

[5] Eric C. R. Hehner. Predicative programming. *Communications of the ACM*, 27(2):134–151, February 1984.

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

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

[8] Leslie Lamport. TLA—temporal logic of actions. A web page, a link to which can be found at URL `http://lamport.org`. The page can also be found by searching the Web for the 21-letter string formed by concatenating `uid` and `lamporttlahomepage`.

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

[10] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[11] Leslie Lamport. Byzantizing paxos. In David Peleg, editor, *Distributed Computing: 25th International Symposium, DISC 2011*, volume 6950 of *Lecture Notes in Computer Science*, pages 211–224. Springer-Verlag, 2011.

[12] 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.

[13] 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.

[14] Eric Verhulst, Raymond T. Boute, José Miguel Sampaio Faria, Bernard H. C. Sputh, and Vitaliy Mezhuyev. *Formal Development of a Network-Centric RTOS*. Springer, New York, 2011.