Chapter 1

Introduction

1.1 Who Am I?

Dear Reader. I am inviting you to spend many pages with me. Before deciding whether to accept my invitation, you may want to know who I am.

I was educated as a mathematician; my doctoral thesis was on partial differential equations. While a student, I worked part-time and summers as a programmer. At that time, almost all programs were what I will call *traditional* programs—ones with a single thread of control that take input, produce output, and stop.

After obtaining my doctorate, I began working on concurrent algorithms—ones comprising multiple independent threads of control, called *processes*, that are executed concurrently. The first concurrent algorithms were meant to be executed on a single computer, with processes communicating through a shared memory. Later came distributed algorithms—concurrent algorithms designed to be executed on multiple computers in which processes communicate by message passing.

This is not the place for modesty. I was very good at concurrency—both writing concurrent algorithms and developing the theory underlying them. The first concurrent algorithm I wrote, published in 1974, is still taught at universities. In 1978 I published what is probably the first paper on the theory of distributed computing. I have received many awards and honors for this work, including a Turing award (generally considered the Nobel prize of computer science) for "fundamental contributions to the theory and practice of distributed and concurrent systems".

1.2 Who Are You?

You probably belong to one of two classes of people who I will call *scientists* and *engineers*. By scientists, I mean computer scientists who are interested in concurrent computing. If you are a scientist, you should be well-prepared to decide if this book interests you and to read it if it does.

By engineers, I mean people involved in building concurrent programs. If you are an engineer, you might have a job title such as programmer, software engineer, or hardware designer. I need to warn you that this book is about a science, not about its practical application. Practice is discussed only to explain the motivation for the science. If you are interested just in using the science, you should read about the language TLA$^+$ and its tools, which are the practical embodiment of the science [28, 35]. But if you want to understand the underlying science, then this book may be for you.

Like many sciences, the book's science of concurrent programs is based on mathematics. The book assumes only that you know the math one learns before entering a university. The basics of all the ordinary math you will need—"ordinary" meaning not peculiar to this science—is explained in Chapter 2. Some additional ordinary mathematics is introduced later as needed. The appendix contains a brief summary of all this math, most of which is taught at universities in an introductory math course for computer science students, though probably not the way it is presented here. Scientists should be used to reading math. You may find the math hard if you're an engineer. But unless mis-education has burdened you with an insurmountable fear of mathematics, I encourage you to give the book a try. Learning the math will improve your thinking.

1.3 The Origin of the Science

The science that is the subject of this book, which I will call *our* science, is a mathematical theory with a practical goal. That goal is to help build concurrent programs that work correctly. Exactly what "working correctly" means and why it's an important goal are explained in Section 1.4. The origin of our science explains how I came to believe it's a good foundation for trying to achieve that goal.

1.3.1 The Origin of the Theory

The first concurrent algorithm was published in 1965 by Edsger Dijkstra [9]. I started writing concurrent algorithms around 1973, and I quickly learned

that they were hard to get right. The many possible orders in which the operations of different processes can be executed leads to an enormous number of possible executions that have to be considered. The only way to ensure that the algorithm worked correctly was to prove that it did.

By the 1970s, a standard approach had been developed for proving correctness of traditional programs. Around 1975, I and a few other computer scientists began extending that approach to concurrent algorithms [4, 24, 29, 45]. Concurrent algorithms were usually written in pseudocode plus some informal explanation of what the pseudocode meant. I came to realize that all these methods for proving correctness could be explained by describing a concurrent algorithm as what I am now calling an *abstract program*; and an abstract program could be described mathematically.

Correctness of an algorithm was expressed by properties required of its executions. I came to realize that correctness can also be expressed by an abstract program—a more abstract, higher-level one than the abstract program describing the algorithm. Proving correctness means showing that the abstract program describing the algorithm implements the abstract program describing its correctness, and I developed a method for doing that.

This work culminated around 1990 with a way to write an abstract program as a single formula [32]. The formula is written in an obscure form of math called temporal logic. The particular temporal logic is called TLA (for the Temporal Logic of Actions). Most of the TLA formula for an abstract program consists of ordinary math that expresses essentially what was described by pseudocode. Temporal logic replaces the informal explanation of the pseudocode. The assertion that one abstract program implements another is expressed as logical implication together with mathematical substitution.

Throughout this period, I was writing correctness proofs of the algorithms I was inventing. This showed me that my way of reasoning with abstract programs worked in practice. However, I discovered that as my algorithms got more and more complicated and the formulas describing them became larger, the method of writing proofs used by mathematicians became unreliable. It could not ensure that all the details were correct. I had to devise a method of hierarchically structuring proofs to keep track of those details.

1.3.2 The Origin of the Practice

I have spent most of my career as a member of industrial research labs. The computer science I have done has been motivated by the problems facing

system builders—sometimes before they were aware of those problems. I have devoted the last part of my career to developing tools to help them—both intellectual tools to help them think better and programs to help them detect errors before they are implemented in code. These tools are based on what I learned by writing and reasoning about concurrent algorithms.

Programming is not just coding. It requires thinking before we code. Writing algorithms taught me that there are two things we need to decide before writing and debugging the code: *what* the program should do, and *how* the program should do it. Most programmers think that the code itself adequately describes "how the program should do it", but I learned that we need a higher-level, more abstract description of what the program does. To emphasize that programming is more than just coding, I will use the name *coding language* for what are commonly called programming languages.

An algorithm is an example of a description of how a program should do something. Concurrent algorithms are hard to understand. To invent them, I had to be able to write them in the simplest way possible. Algorithms were usually written in pseudocode to avoid the complexity that real code requires for efficient execution. I developed a way to describe concurrent algorithms in math that was more precise and no harder to understand than pseudocode.

Engineers who build complex systems usually recognize the need for describing what their programs do in a simpler, more abstract way than with code. I decided that abstract programs written in math provided such a way for describing the aspects of a system that involve concurrency. By about 1995, I had designed a complete language called TLA$^+$ that engineers could use to write abstract programs in TLA.

The abstract programs I know of that have been written by engineers to describe what a system should do generally consist of about 200–2000 lines of TLA$^+$. All but a few of those lines are ordinary math, not temporal logic. As with code, those formulas are made easy to understand by decomposing them into smaller pieces. This is done using simple definitions, rather than the more complex constructs of coding languages.

To formalize mathematics and make it easier to write long formulas, I had to add to TLA$^+$ some concepts and syntax not present in the math commonly used by mathematicians—for example, variable declarations, grouping definitions into modules, and notation for substitution. This book uses TLA, but not TLA$^+$, because the examples with which it illustrates our science are short and simple.

The kind of hierarchically structured proofs I devised can also be written in TLA$^+$, and there is a program for checking the correctness of those proofs.

However, with today's proof-checking technology, writing machine-checked proofs takes more time than engineers generally have. By the time I designed TLA$^+$, model checking had become a practical tool for checking the correctness of abstract programs and was often used by engineers. A model checker can essentially check correctness of all possible executions of a very small instance of an abstract program. This turns out to be very effective at detecting errors. There are two model checkers for abstract programs written in TLA$^+$, using two complementary approaches.

A program's code can, in principle, be described by a (concrete) abstract program and could, in principle, be written as a TLA$^+$ formula. For a simple program (or part of a program), the code can be hand-translated to TLA$^+$ and checked with the TLA$^+$ tools. Usually, the length of the program and the complexity of the coding language makes this impractical.

From the point of view of our science, it makes no difference how long the formula describing an abstract program is. We therefore consider a program written in a coding language to be an abstract program. And since we are considering only abstract programs, we will let *program* mean abstract program. We will call an (abstract) program written in code a *concrete* program.

Although we don't write them as formulas, viewing concrete programs as (abstract) programs provides a new way of thinking about them. One benefit of this way of thinking is that understanding what it means for a concrete program to implement a higher-level abstract program can help avoid coding errors.

1.4 Correctness

Thus far, our science has been described as helping to build concurrent programs that work correctly. Working correctly is a vague concept. Here is precisely what it is taken to mean in this book.

We define a *behavioral property* to be a condition that is or is not satisfied by an individual execution of a program. For example, *termination* is a behavioral property; an execution either terminates or else it doesn't terminate, meaning that it keeps executing forever. We say that a program satisfies a behavioral property if every possible execution of the program satisfies it. A program is considered to *work correctly*, or simply to *be correct*, if it satisfies its desired behavioral properties.

That every possible execution of a program satisfies its behavioral properties may seem like an unreasonably strong requirement. I would be happy

if a program that I use does the right thing 99.99% of the times I run it. For many programs, extensive testing can ensure that it does. But it can't for most concurrent programs. What a concurrent program does can depend on the relative order in which operations by different processes are executed. This makes the program nondeterministic, meaning that different executions can do different things, even if the program receives identical inputs. This can result in an enormous number of possible executions, and testing can examine only a tiny fraction of them. Moreover, a concurrent program that has run correctly for years can start producing frequent errors because a small change to the computer hardware, the operating system, or even the other programs running at the same time causes incorrect executions that have never occurred before. The only way to prevent this is to ensure that every possible execution satisfies the behavioral properties.

Model checking is more effective at finding errors in concurrent programs than ordinary testing because it checks all possible executions. However, it does this only on a few small instances of the program—for example, an instance with few processes or one that allows only a small number of messages to be in transit at any time.[1] Engineering judgment is required to decide if correctness of those instances provides enough confidence in the correctness of the program.

There is one way testing could find errors in concrete programs. When building a concurrent system, an abstract program is often used to model how the processes interact with one another, and the correctness of that program is checked. The concrete program is then coded by implementing each process of the more abstract program by a separate process in the code. Since there is no concurrency within an individual process, testing that the concrete program implements the more abstract program has a good chance of finding coding errors. Research on this approach is in progress.

1.5 A Preview

To give you an idea of what our science is like, this section describes informally a simple abstract program for Euclid's algorithm—a traditional algorithm that computes a value and stops. It's a very simple concurrent program in which the number of processes equals 1. Our science applies to single-process programs, although there are simpler sciences that work quite

[1] There are techniques for proving the correctness of a program by model checking a simpler program, but they have not been implemented for abstract programs written in TLA$^+$.

well for them.

Euclid's algorithm computes the greatest common divisor (GCD) of two positive integers that we will call M and N. For example, the GCD of 12 and 16, written $GCD(12, 16)$, equals 4 because 4 is the largest integer such that 12 and 16 are both multiples of that integer. The algorithm is an abstract program containing two variables that we name x and y. Here is its prose description.

> Start with x equal to M and y equal to N and repeatedly perform the following action until the program stops:
>
> > If the values of variables x and y are equal, then stop; otherwise, subtract from the variable having the larger value the value of the other variable.[2]
>
> When the program has stopped, x and y equal $GCD(M, N)$.

I believe most engineers and many scientists can't explain why an execution of Euclid's algorithm computes $GCD(M, N)$, which means that they don't understand the algorithm. Here is the explanation provided by our science, beginning with how we view executions.

We consider an execution to be a sequence of states. For Euclid's algorithm, a state is an assignment of values to the program variables x and y. We write the state that assigns 7 to x and 42 to y as $[x :: 7,\ y :: 42]$. Here is the sequence of states that is the execution of Euclid's algorithm for $M = 12$ and $N = 16$.

$$\begin{bmatrix} x :: 12 \\ y :: 16 \end{bmatrix} \rightarrow \begin{bmatrix} x :: 12 \\ y :: 4 \end{bmatrix} \rightarrow \begin{bmatrix} x :: 8 \\ y :: 4 \end{bmatrix} \rightarrow \begin{bmatrix} x :: 4 \\ y :: 4 \end{bmatrix}$$

The states in the sequence are separated with arrows because we naturally think of an execution going from one state to the next. But in terms of our science, the algorithm and its execution just *are*; they don't go anywhere.

What an algorithm does in the future depends on its current state, not on what happened in the past. This means that in the final state of the execution, in which x and y are equal, they equal $GCD(M, N)$ because of some property that is true of every state of the execution. To understand Euclid's algorithm, we must know what that property is.

That property is $GCD(x, y) = GCD(M, N)$. (Chapter 3 explains how we show that every state satisfies this property.) Because an execution

[2] You may have seen a more efficient modern version of Euclid's algorithm that replaces the larger of x and y by the remainder when it is divided by the smaller. For the purpose of this example, it makes little difference which we use.

stops only when x and y are equal, and $GCD(i, i)$ equals i for any positive integer i, this property implies that x and y equal $GCD(M, N)$ in the final state of the execution.

That the formula $GCD(x, y) = GCD(M, N)$ is true in every state of a program's execution is a behavioral property. A behavioral property that asserts a formula is true in all states of an execution is called an *invariance* property, and the formula is called an *invariant* of the program. Correctness of any concurrent program depends on it satisfying an invariance property. To understand why the program is correct, we have to know the invariant of the program that explains its correctness.

The invariant $GCD(x, y) = GCD(M, N)$ shows that, if Euclid's algorithm terminates, then it produces the correct output. A traditional program must also satisfy the behavioral property of termination. The two behavioral properties

- The program produces correct output if it terminates.
- The program terminates.

are special cases of the following two classes of behavioral properties that can be required of a concurrent program:

Safety What the program is allowed to do.

Liveness What the program must eventually do.

These two classes of properties are defined precisely in Section 4.1. Termination is the only liveness property required of a traditional program. There are many kinds of liveness properties that can be required of concurrent programs.

Euclid's algorithm satisfies its safety requirement (being allowed to terminate only if x and y equal $GCD(M, N)$) because the only thing it is allowed to do is start with $x = M$ and $y = N$ and execute its action. That is, it satisfies its safety requirement because it is assumed to satisfy the safety property of doing only what the description of the algorithm allows it to do.

Euclid's algorithm satisfies its liveness requirement (eventually terminating) because it is assumed to satisfy the liveness property of eventually performing any action that its description allows it to perform. (Section 3.4.2.8 shows how we prove that the algorithm terminates.)

I have found it best to describe and reason about safety and liveness in different ways. In our science, temporal logic plays almost no role in

handling safety, but it is central to handling liveness. The TLA formula for an abstract program is the logical conjunction of a safety property and a liveness property.

A single-process algorithm that computes a value and stops doesn't seem to be a good example for a science of concurrent programs. So, let's consider a concurrent version of Euclid's algorithm. It's a two-process version of the algorithm, suitable for execution on a single computer with the processes communicating through shared variables. We call the two processes the x process and the y process. The algorithm uses the same program variables as the one-process version of Euclid's algorithm and it begins in the same starting state, with $x = M$ and $y = N$. If the x process hasn't stopped, it is allowed to execute the following action whenever the *when* condition is true:

When $x \geq y$, stop if $x = y$, else subtract the value of y from x.

Process y is the same as process x, except with x and y interchanged.

Written in pseudocode, this version of Euclid's algorithm looks different from the one-process version. However, if we consider the executions of the two versions, we see that they are the same. That is, they have the same sequence of states, where a state is an assignment of values to x and y. Whether we view Euclid's algorithm as a one- or two-process algorithm may affect how we implement it with a concrete program. An implementation of the two-process algorithm with a two-process concrete program would probably be less efficient than a single-process implementation of the one-process algorithm. But since these two versions of the algorithm have the same executions, from the point of view of correctness they are the same algorithm. Both versions are written as the same TLA formula. More precisely, their formulas are equivalent. There are many equivalent ways to write a mathematical formula. How we choose to write the TLA formula for Euclid's algorithm can depend on whether we view it as a one- or two-process algorithm.

1.6 Why Math?

The science of bridge building has a mathematical basis, but bridge designers don't represent a bridge by a mathematical formula. Why should we describe an abstract program with one? The simple answer is, because we can. A concrete program is not a physical object; it's a concept. Code is just one representation of that concept. While possible in theory, writing a

mathematical representation of a concrete program is not practical. However, for simpler abstract programs, it is possible; and I've found it to be a good way to represent them.

Math has been developed over thousands of years to be simple and expressive. An abstract program ignores many implementation details, which often means allowing multiple possible implementations. This is simple to express in math. Code is designed to describe one way of computing something, and it can be hard or even impossible to write code that allows all those possibilities. Even pseudocode, being based on concepts from coding languages, lacks the simplicity and expressiveness of math.

One place we want to allow many possible implementations is in describing what the program's environment can do. A program can't work in an arbitrary environment. An implementation of Euclid's algorithm will not produce the correct answer if the operating system can arbitrarily modify the variables x and y. A concurrent program can interact with its environment in complicated ways, and we have to state explicitly what the program assumes about its environment to know if it's correct. We usually want to assume as little as necessary about the environment, which means the abstract program should allow it to have many different behaviors.

Unanticipated behavior of the environment is a serious source of errors in concurrent programs. Part of the environment of a program is likely to be another program, such as an operating system. Avoiding errors may require finding answers to subtle questions about exactly what that other program does. This is often difficult because the only description of what it does, other than its code, is likely to be imprecise prose. When writing the abstract program to describe what our concrete program does, describing what the environment can do will tell us what questions we have to ask.

The expressiveness of math, embodied in TLA, provides a practical method of writing and checking the correctness of high-level designs of systems. Such checking can catch errors early, when they are easier to correct. TLA$^+$ is used by a number of companies, including Amazon [42], Microsoft, and Oracle. Math also provides a new way of thinking about programs that can lead to better programming. While there is usually no way to quantify the result of better thinking, it was possible in the following instance.

Virtuoso was a real-time operating system. It controlled some instruments on the European Space Agency's Rosetta spacecraft that explored a comet. Its creators decided to build the next version from scratch, and they started by writing a high-level design in TLA$^+$. They described their experience in a book [49]. The head of the project, Eric Verhulst, wrote this in an email to me: